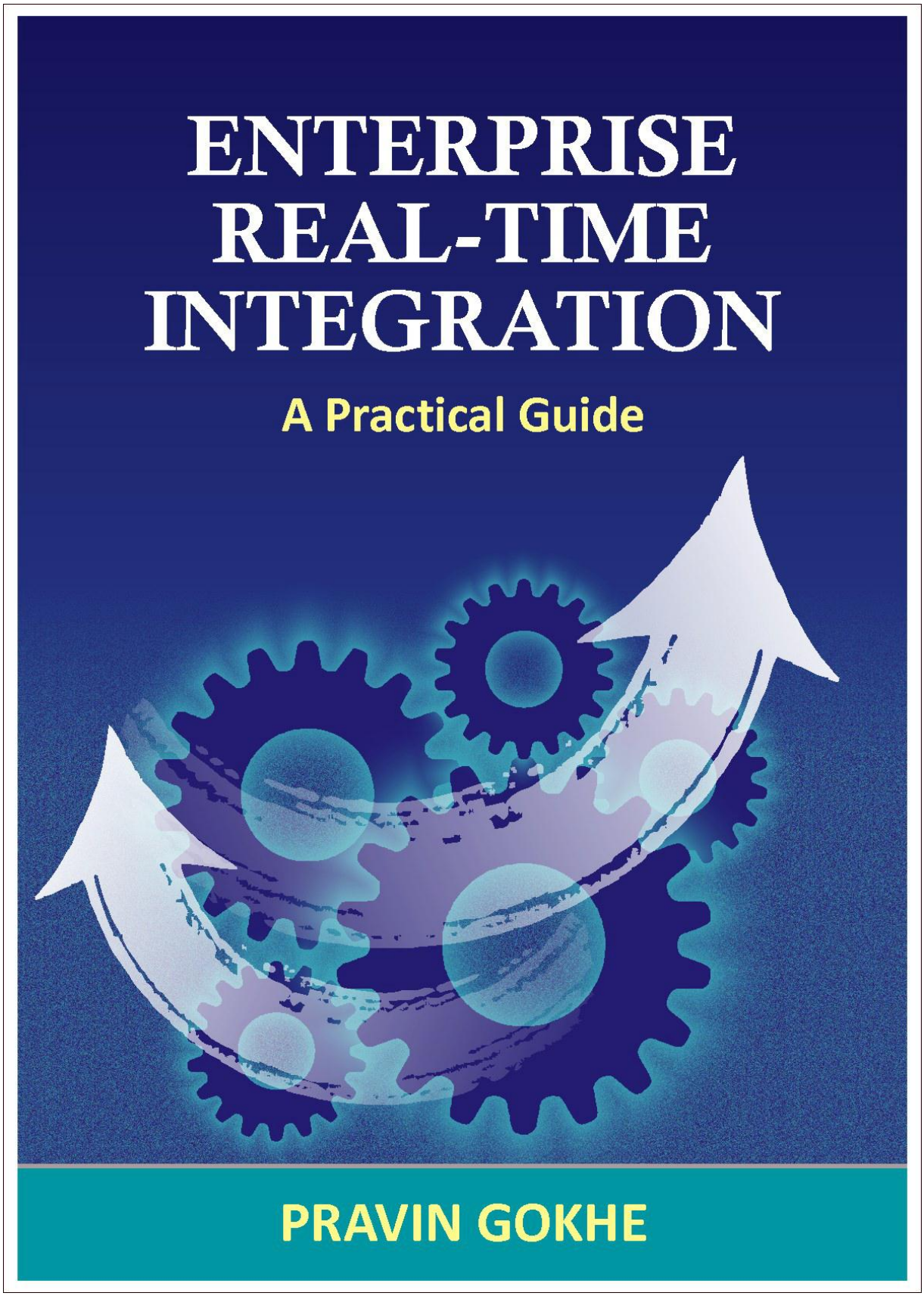
ENTERPRISE
REAL-TIME
INTEGRATION
A Practical Guide
PRAVIN GOKHE

Copyright

Dedicated To

My father: R. S. Gokhe

Acknowledgements

Writing this book was a long journey. I would like to express my sincere gratitude to many people who have been associated with me through this book.

First of all I would like to thank my wife - Bhumika, my daughter – Srishti and my mother - Shakuntala, who have been always with me during this journey. The release of this book would not have been possible without the support, love and encouragement I received from them.

Special and wholehearted thanks to AVRS Rajesh and Shiju Nair for taking out their personal time to review this book thoroughly and providing valuable feedback. Review work would not have been possible without their timely support and dedication.

Thank to my seniors – Parveen Malhotra, Atul Kahate, Dr. Shailesh and Yogesh Sinhal who have been constant source of inspiration for me.

I would also like to thank my colleagues Anand, Shabber, Sumesh, Krishna, Mickey and Sandesh for helping me with the review of some specific content in the book.

Preface

When it comes to articulating the vision of real-time integration architecture for Enterprise as a whole, it is very important to have the right understanding of various prevalent integration use cases - Messaging, SOA-EAI, APIs and B2B integration, and synergies among respective implementation platforms. Most of the times I have noticed that the teams working on integration initiatives/programs often miss the broader perspective and strategic outlook on integration; teams deal with integration as a vanilla mechanism to connect two applications or two Organizations. This eventually leads to suboptimal design, complex integration landscape and non-scalable environment.

These challenges motivated me to write this book which is a blend of Business and Technology outlook, and deals with integration topic from a broader perspective. The integration space is always about right strategies and optimal design thinking; the strategic perspective while integrating Enterprise capabilities (functionalities and data) and partners-third parties can be a real enabler for Business.

I have been working in integration space for almost 15 years now and have worked on many key Enterprise-wide strategic initiatives related to SOA-EAI, API management, B2B integration and Messaging. While working on these large-scale initiatives, I have learnt many practical tactics and approaches that can help in building seamless and scalable integration solutions. My objective through this book is to share the practical knowledge I have gained over the years and provide an insight into real-time integration while promoting architecture and design thinking.

Understanding various dimensions of real-time integration should provide one with the ability to define strategic and sustainable integration solution for a given use case, which otherwise is a very common area for oversight and things can go wrong leading to unmanageable and complex integration ecosystem.

Intended Audience

The book covers various aspects of real-time integration from broader perspective of Enterprise, Business and Technology. Since this book takes a unique approach of explaining basic and advanced concepts with real-world examples-scenarios, even the IT professionals with little background on integration can use this book to get an insight into real-time integration space. Existing practitioners in integration space will be able to significantly boost their capability to define the right and scalable integration solutions while clearly understanding synergies among various integration platforms (Messaging, EAI-SOA, API Management and B2B integration).

List of audience who will be benefited by reading this book

- Integration Architects
- Enterprise Architects
- Integration Designers/Engineers
- Integration Tech Leads
- Integration Development Leads
- Integration Developers
- Technical Managers

Unit I

Messaging

Contents

Introduction: Messaging

This unit is divided into 3 chapters

- Messaging Overview
- Messaging System Core Concepts
- Messaging System Advanced Concepts

Objective: The objective of this unit is to

- Explain the need of Messaging system and outline its role in integrating various heterogeneous applications within Enterprise.
- Describe the various players in overall messaging architecture, fundamental messaging styles and their purpose, features of Messaging system – Destination, avoiding data loss, message selector and so on.
- Define '*Enterprise Real-time Integration Reference Architecture: Messaging*'.
- Outline key strategies for building federated messaging architecture for better scalability and reduced Business impact.
- Explain advanced features of messaging, design considerations and anti-patterns.
- Discuss some of the non-functional requirements such as Fault Tolerance and Monitoring.

Core as well as advanced concepts are demonstrated using scenarios and case studies from various domains – Banking, Telecom and Supply Chain.

Messaging system plays a key role in integrating Enterprise application. Obviously, the perspective will not be complete without unfolding the need of messaging in the first place. So the first chapter starts with explaining the IT evolution, the heterogeneous landscape and the need of Messaging system.

During the explanation of concepts in the book, the important points and keywords are highlighted using maroon color.

Acronyms

JMS	Java Message Service
MOM	Message-Oriented Middleware
CRM	Customer Relationship Management
ERP	Enterprise Resource Planning
MDM	Master Data Management
BFSI	Banking Financial Services and Insurance

Chapter 1: Messaging Overview

- IT evolution and heterogeneous IT application landscape
- Traditional Integration Scenario
- Need of Message-Oriented Middleware (MOM)
- Introduction to Messaging
- Benefits of Messaging System

IT evolution and heterogeneous IT application landscape

The programming languages or platforms available for developing business applications several decades ago were very limited. COBOL was one of the mainstream programming languages available that time for building business applications which used to run on large computers such as mainframe. These business applications (such as Payroll, Core Banking, Billing, Accounting etc.) written in COBOL are still in use today. These applications built using COBOL programs and running on mainframe are commonly referred as legacy applications.

Programming languages and the technology space for building business applications have been constantly evolving. After mainframe-COBOL, came the era of packaged business applications - applications supporting the business functions out of the box. So with packaged applications, there was no need to build business applications from scratch, Organizations can buy these products from vendor, customize to their requirements and enable business functions quickly.

Then with the advent of web technologies, java became the mainstream programming language enabling faster development of web based applications, self-service portals and so on.

Mainframe-COBOL, packaged applications and java are just some examples from the vast landscape of platforms deployed in any Organization.

The application landscape in any Organization keeps evolving over the period of time to offer better functionalities and features. For example, application might be incapable to support new functionalities-features, resulting in Business losing market shares to the competitors. In such circumstances, it becomes essential to replace such application with newer platform/technology. So it could be

- An adoption of better ERP packaged application to increase operational efficiency.
- An upgrade of CRM application to empower Sales, Marketing and Customer Service agents to server customer better.
- New implementation of Master Data Management (MDM) solution ensuring quality data about Customer, Product and so on.

However, in some cases, the application would have been customized over the period of decades and it becomes nearly impossible to replace it with new platform/technology due to complexity and risk to the Business. Additionally, there may not be need to replace it as application might be stable enough and supporting all the functionalities required by the Business.

Going by general tendency of Organizations, adoption to technology-packaged application happens in piecemeal by choosing best-of-breed products available from various vendors.

So over the period of time, this journey of IT automation leaves Organization with heterogeneous application landscape based on various technologies and platforms.

To understand this better, let's take an example of Consumer Electronics manufacturing company, ABC. Let's assume that over the period of time Organization have landed into application landscape as highlighted below

- Call Center application – **from CRM vendor**
- Order Management (OM) application – **from ERP Vendor 1**
- Order Fulfillment (OF) and Shipping application – **from ERP Vendor 2**
- Billing and Invoicing application – **Custom Java application**
- Accounting System – **COBOL application on mainframe**

Now let's take a look at the simplistic view of Order-to-Cash Business Process happening within the ABC Organization to process Dealer's order request for products:

Step 1 - Customer Service agent captures dealer's order through Call Center CRM application and submits the same for processing.

Step 2 - Order Management ERP application checks dealer's available credit limit, checks availability of items and confirms the order.

Step 3 - Order Fulfillment and Shipping ERP application executes Pick-Pack-Ship and sends advanced shipment notification (ASN) to the dealer.

Step 4 - Billing and Invoicing Java application generates invoice for the items shipped and sends it to the dealer.

Step 5 - Accounting entry corresponding to the invoice is posted in legacy Accounting system (recording receivables). Once the payment is received from the dealer, remittance is applied against the account receivable.

The diagram below depicts the key tasks in Order-to-Cash Business Process.

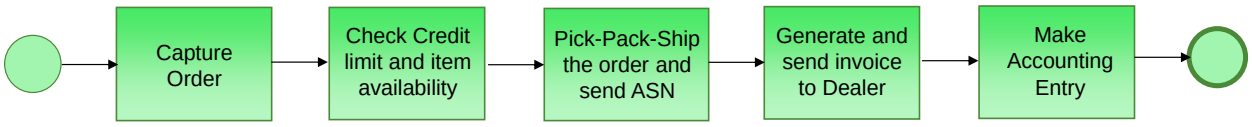

Figure: Business Process – Order-to-Cash

** Please note that the actual process may be more complex involving collection, Reconciliation application etc. However, we will focus on the simplistic view of process as the objective is to understand heterogeneous application landscape.*

The diagram below depicts the various technology platforms-applications involved in realizing Order-to-Cash Business Process.

Figure: Platforms supporting Order-to-Cash Business Process

As very evident from the above diagram, various heterogeneous platforms-technologies are involved in end-to-end Order-to-Cash Business Process and they need to interface with each other in real-time basis to successfully complete the overall Order-to-Cash Business Process.

Traditional Integration Scenario

To achieve the real-time integration between the Enterprise applications of ABC, if the traditional approach of direct point-to-point connectivity between the applications is followed, then it creates the tight coupling between them resulting in more and more dependency. The diagram below shows the tight coupling between the various interfacing applications, with applications adopting each other's proprietary protocols, semantics and formats.

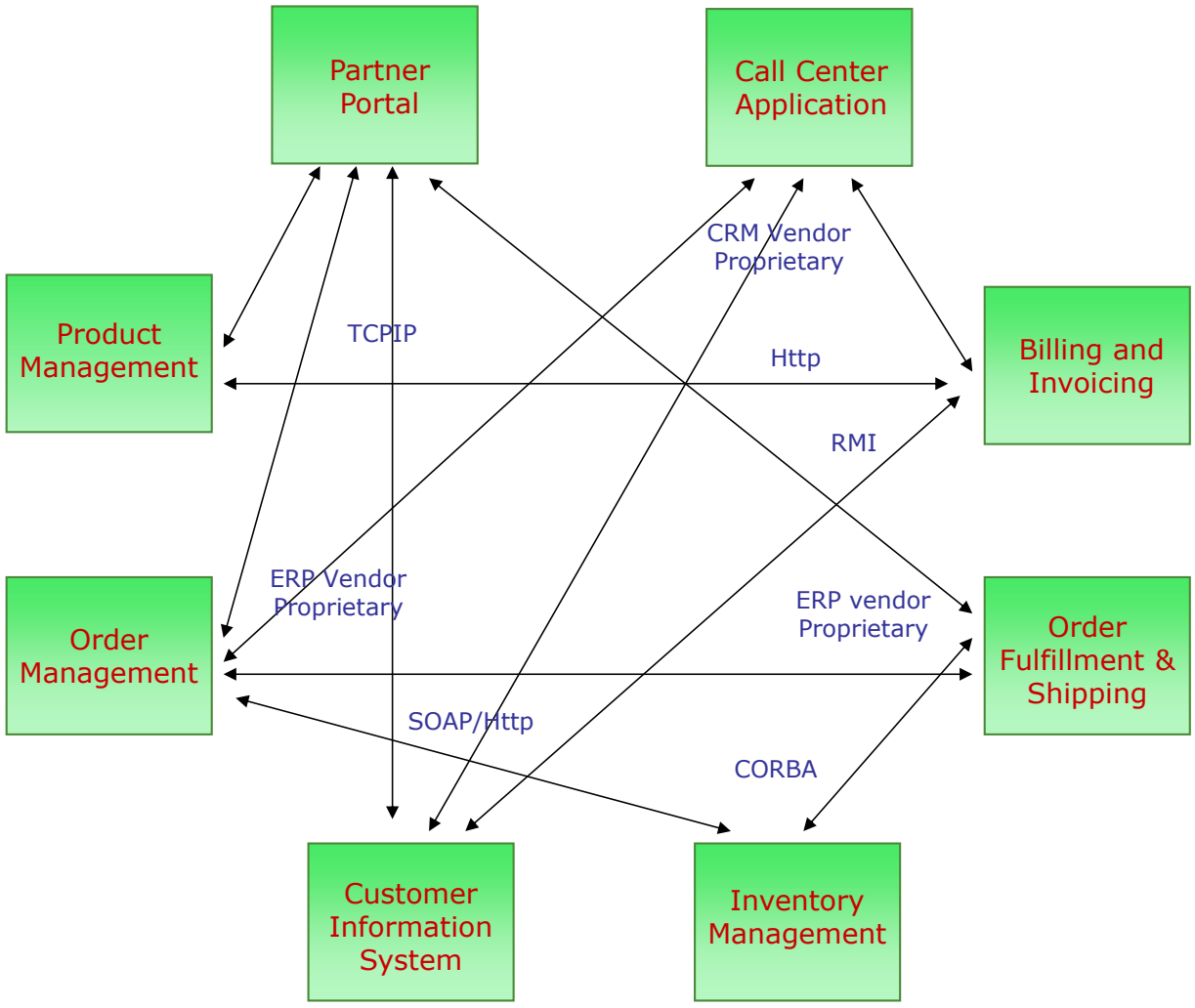

Figure: Heterogeneous application landscape with point-to-point connectivity

Now let's consider the scenario in future where the Order Management ERP application is being replaced with ERP application from another vendor; this could be because the new ERP application is offering better functionality and operational efficiency by means of enhanced process control and visibility into processes. All the applications (Self Service Portal, CRM, Inventory Management, Order Fulfillment and Shipping) interfacing with earlier ERP application in a proprietary way will be impacted and these application would require changes to adopt to new ERP application's protocol, format and semantic. Such kind of integration pattern, where the modernization or upgrade of one particular application impacts all other interfacing applications is not desirable as it involves additional effort, cost and time. As a matter of fact, it becomes completely unmanageable and costly affair to deal with the changes. The diagram below depicts how change in one particular ERP application impacts other interfacing applications (highlighted in red color).

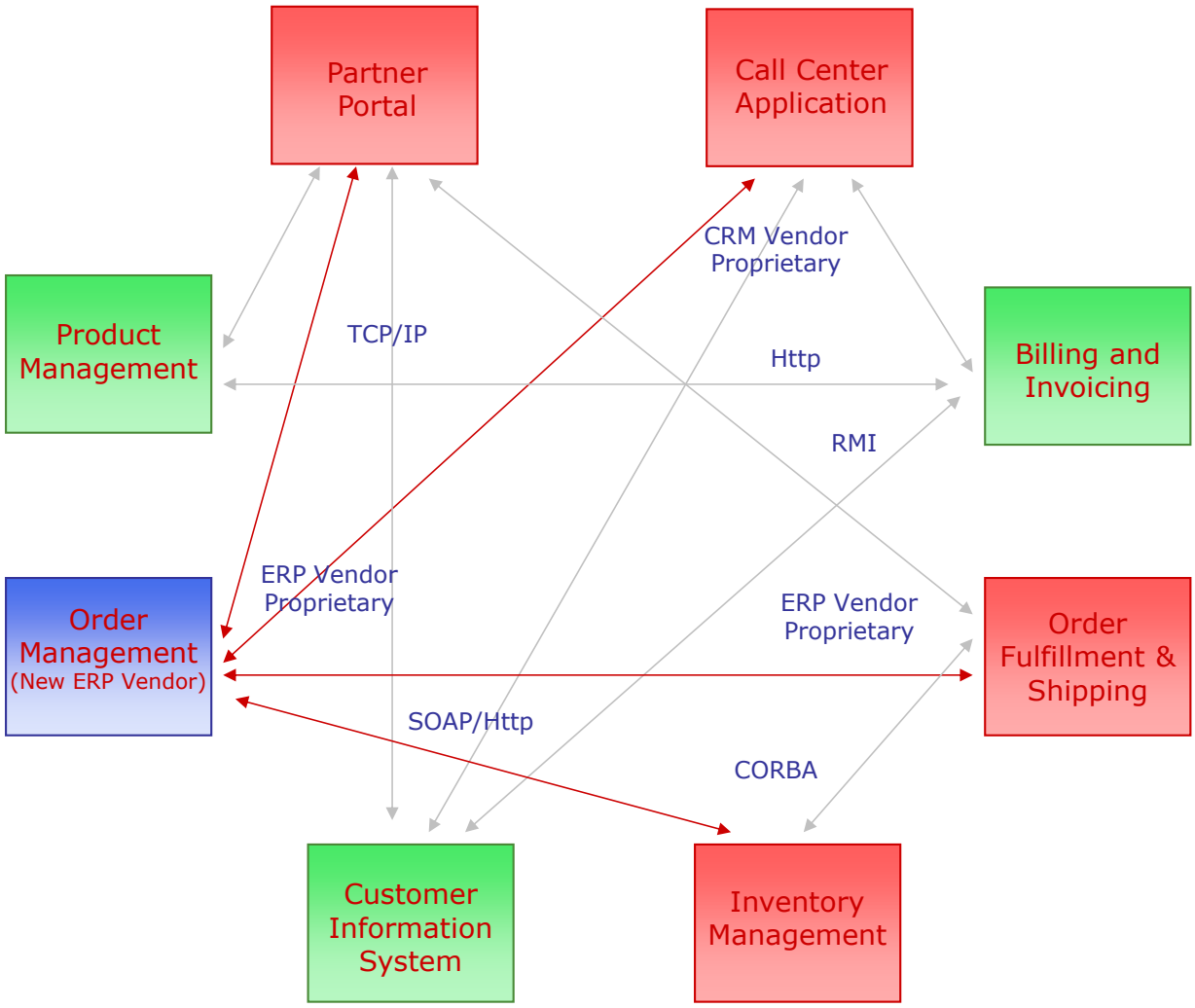

Figure: Impact of Application Upgrade

To summarize, direct point-to-point application integration:

- Creates tight coupling between interfacing applications, as application adopt each other's protocol, format and data semantic.
- Creates larger impact on interfacing applications in case of application upgrade or technology modernization.
- Can adversely impact application's Business Processes, if the target application with which it integrates is temporarily not available due to some reason.

So what is the way to reduce the impact and decouple these Enterprise applications in terms of protocol, format and semantic? And at the same time, these applications should be able to integrate with each other seamlessly to ensure successful execution of end-to-end Business Processes. We will look at all these aspects gradually. However, let's start with addressing protocol issue first and creating a loosely coupled environment.

Need of Message-Oriented Middleware (MOM)

Challenges with point-to-point application connectivity pattern creates the need for Message-Oriented Middleware (MOM). MOM provides the loose coupling among Enterprise applications by means of common standard and protocol. The diagram below depicts how introducing Messaging can isolate applications from each other. This is the first level of integration solution Organizations must adopt to bring-in the loose coupling while integrating Enterprise applications.

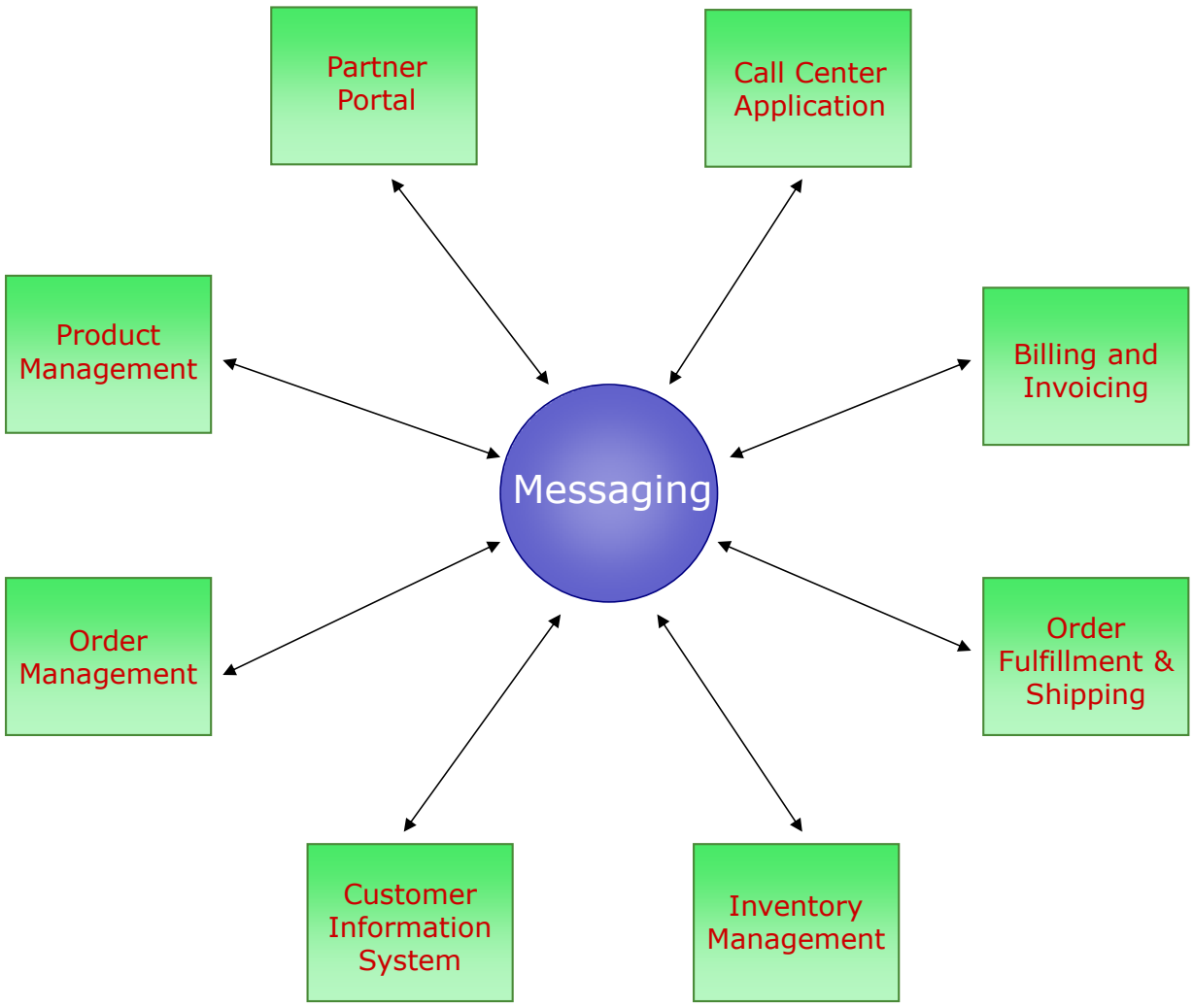

Figure: Message-Oriented Middleware

As shown in the above diagram, MOM provides the decoupling between applications. Application can integrate with other applications in real-time without knowing which

application at the other end is providing the desired functionality or data. Application can change without impacting other interfacing applications. Applications can operate in a plug-and-play mode by using Messaging as a backbone; new application can be quickly integrated with rest of the Enterprise applications without even knowing their protocol, location, platform and technology. New application just needs to know the interfacing mechanism with the Messaging system and that's all.

Messaging solution also acts as a foundation for rest of the advanced real-time integration patterns and solution Organization needs eventually.

In the subsequent section, we will focus on Messaging Concept in detail.

Introduction to Messaging

Messaging system or so called Message-Oriented Middleware acts as a backbone for integrating enterprise applications in real-time. It facilitates integration of disparate business applications in a reliable manner.

In a simplest form, Messaging system allows any application to send a message to a central messaging server and allows another application to receive a message from the messaging server.

The diagram below depicts the most basic scenario of integrating 2 applications.

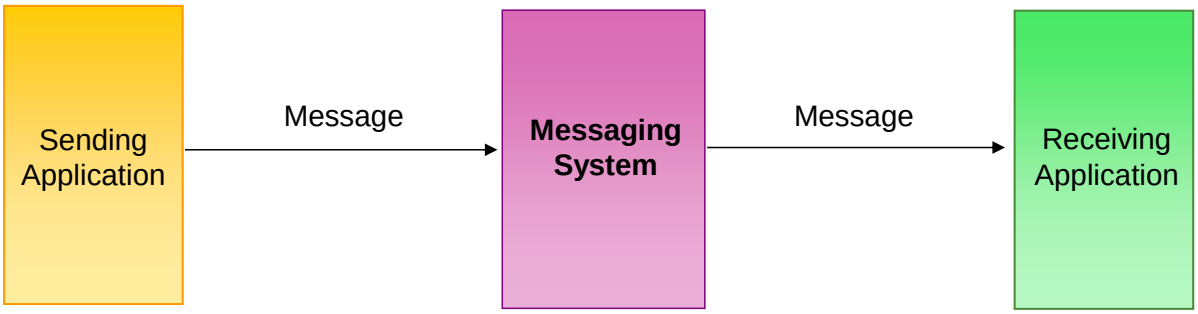

Figure: Message-Oriented Middleware with Sender and Receiver

One of the common misconceptions is that the Messaging systems are primarily used for asynchronous fire-and-forget events. The fact is, Messaging systems can also be used for real-time request-reply scenarios effectively; it is just that the underlying technical mechanism to achieve the request-replay behavior is asynchronous. This will be evident as we deep dive into industry examples in subsequent chapters.

The diagram below depicts Messaging system in the context of Order-To-Cash Process. The Order message is being sent by Call Center CRM application to Messaging system and it is being received by Order Management ERP application to process it further.

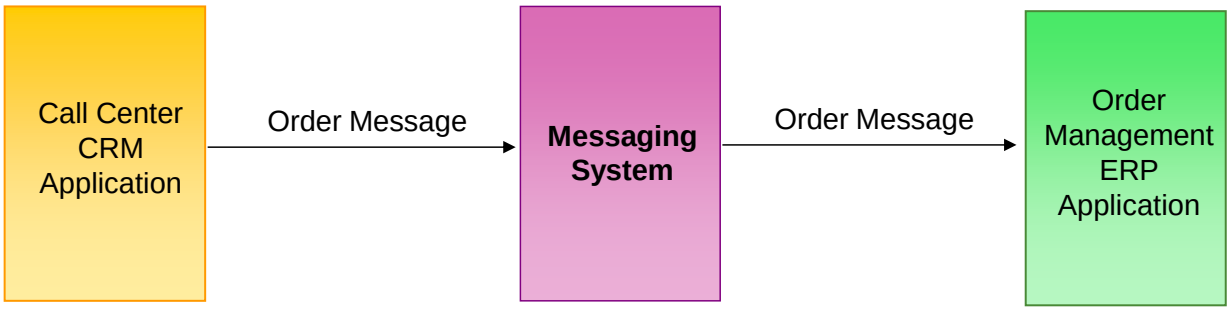

Figure: Message-Oriented Middleware in the context of Order-To-Cash Process

As discussed in the case of ABC Organization, application and technology landscape can be very much diversified - CRM, ERP, Java and so on. While each application is critical for its own functional area, the exchange of information between these applications in a reliable way is also equally important for overall Business Process to execute successfully (fulfilling the Order). Messaging system provides the common facilities to these applications for exchanging data with each other in a reliable way.

Now let's take a quick look at key entities involved in Message-Oriented Middleware

Message:
A Message is the data that the sending application sends to the receiving application. For example, 'Order Request' message sent by CRM application to Order Management application. The structure of the message could be any acceptable format as agreed by source and target system. It could be fixed width, delimited, XML or SOAP.

Client:
Client is the application that sends or receives the messages from Messaging system. The client who creates and sends the message is known as the Message Producer or Sender and the client who receives the message is known as the Message Consumer or Receiver.

Destination:
Destination is at the core of Messaging system. Destination is an object in Messaging system, that

- Represents the target for the message from Message Producer perspective
- Represents the source of the message from Message Consumer perspective

There are 2 types of destination; they have a specific purpose when they should be used

- Queue
- Topic

Destinations will be discussed in detail in the subsequent chapter.

Messaging System (MOM):
Messaging system acts a central server for Message Producer and Message Consumer to exchange messages. Messaging system provides host of services for each message depending upon what attributes have been chosen for that particular message. For example, if producer sends message with *DeliveryMode* property set as "PERSISTENT" to the Messaging system, then the Messaging system offers a service where message is delivered to a Message Consumer in a guaranteed manner, even in case of Messaging system failure. Similarly, if Message Producer wants message not be delivered to consumer if message is not consumed within 15 sec by consumer, then producer can set *time-to-live* property accordingly. Messaging system will ensure message gets deleted from destination, if not consumed by consumer within 15 seconds. These are just couple of examples; more properties and behavior of Messaging system will be discussed in the subsequent chapter.

Client API:
Messaging system provides standard client APIs for sending and receiving applications to interact with Messaging system. Client APIs are standard set of interface-libraries which facilitates creating connection to the Messaging system, creating session, looking up the destination (topic or queue) for sending and receiving messages and so on.

The client APIs are typically available in different programming languages such as C, C++, java, COBOL, C#. This allows heterogeneous applications based on different platform and programming languages to send or receive messages from Messaging system. This is one of the important facilities provided by Messaging system vendors to connect heterogeneous applications to Messaging system seamlessly.

Store:
Messaging system can use either file based store or database instance to store the PERSISTENT messages.

The diagram below depicts all the key entities discussed above.

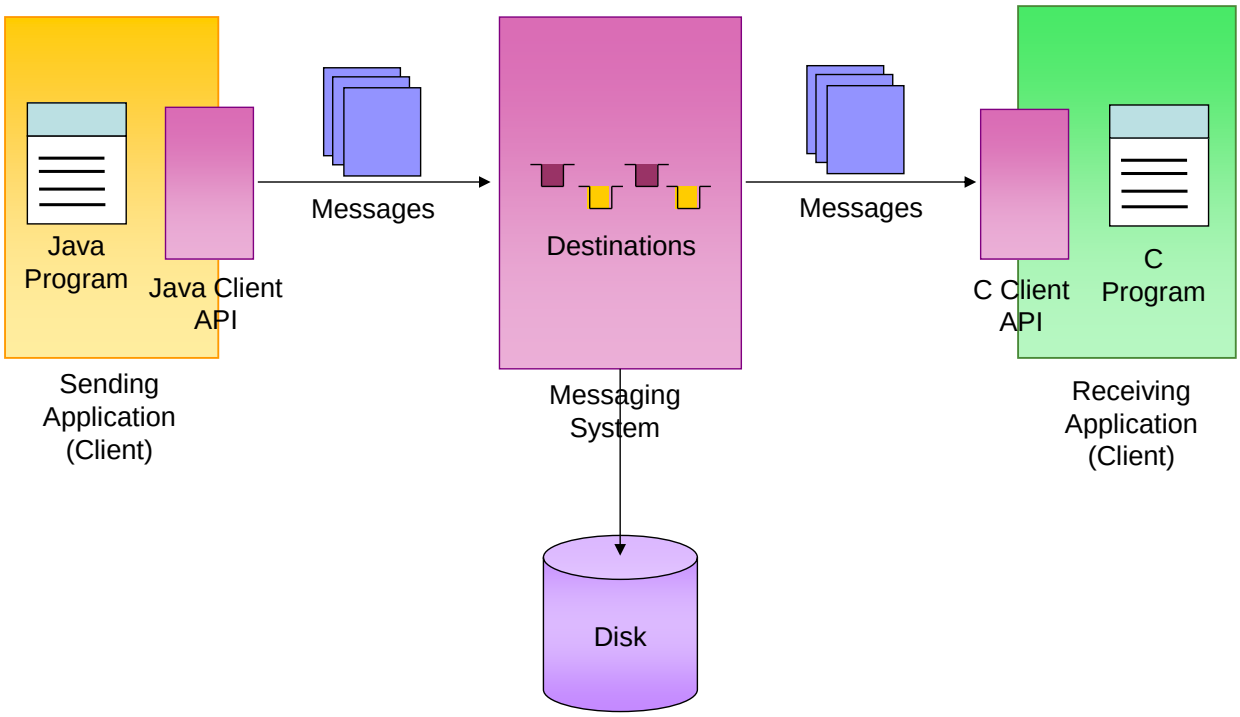

Figure: Messaging System Key Entities

Benefits of Messaging System

Some of the important features provided by Messaging system are - Reliability, Guaranteed Delivery, Loose Coupling and Scalability.

Loose Coupling between Heterogeneous applications (make application accessible to each other without knowing the protocol/language) –
Messaging system bridges proprietary protocol/programming language gap by providing the common messaging protocol to interfacing applications. For example, Message Producer Java application can connect to Messaging system and send the message. The Message Consumer in a different programming language such as C or .Net can consume the same message from Messaging system. This provides the technology independence to interfacing applications and also loose coupling.

Less or no impact during application/technology upgrade –
Technology innovation is an inevitable process in any organization; applications and technology need to be upgraded for better features, customer experience etc. And this all should happen without impacting the existing interfaces with application. Loose coupling provided by Messaging system plays a significant role by isolating applications and thus reducing impact.

Reliability –
It is possible that sometimes receiving application may be down temporarily due to planned maintenance activity. However, messages/data sent by sender application during this time period must reach the target application. Messaging system by default provides the feature of retaining the messages and delivering them to target when the target system comes up.

For example, let's assume a scenario where customer places the request for laptop through e-retailer's online Portal. If the back-end Order Management application is temporarily unavailable due to unforeseen event, it may not be good idea to error out and request customer to re-raise the request. If middleware is being used to integrate online Portal application and back-end Order Management application, *Order* message will be received and retained in Messaging system for delivery to Order Management application without customer being needed to re-raise the request. The *Order* message will be delivered to back-end Order Management application whenever it becomes available for processing. This enhances customer's experience. These types of messages are typically called 'fire-and-forget' events (in this case from Portal application to Order Processing application). On the other hand, once the Portal application sends *Order* message to Messaging system, Portal application can continue processing other tasks without worrying whether the message has reached Order Management application or not. (Or whether Order Management application is available to take request or not). Portal application can rely on Messaging system to take care of delivering *Order Request* to Order Management application in a guaranteed manner and reliably.

Enterprise scalability
Messaging system also provides enterprise level scalability from the perspective that multiple Messaging servers can connect and exchange messages with each other; thus ultimately integrating hundreds of Enterprise applications spread across geographical locations. Messaging can connect one region's application with other region's application seamlessly through various kinds of topologies.

Scalability by design will be discussed in detail in Federated Architecture section.

Chapter 2: Messaging System Core Concepts

- Introduction to JMS
- JMS Messaging Styles
- Message Structure
- Destination
- Avoiding Data Loss
- Message Selector
- Message Compression
- Message Consumption Mechanism
- Exclusive Consumer
- Administration
- Security
- Messaging Platform Component Topology
- Enterprise Real-time Integration Reference Architecture: Messaging
    - Point-To-Point Scenario
    - Publish-Subscribe Scenario

Introduction to JMS

Java Message Service (JMS) is a specification for messaging between applications. It provides a common set of interfaces for Java programs to access Messaging system's features for exchanging data. In a nut shell, JMS brings standardization to which Messaging system needs to adhere to, offering consistent features.

Let's draw the analogy between MOM entities discussed in the earlier section with entities involved in JMS. JMS comprises of the following components:

JMS Provider - JMS Provider is nothing but the Messaging system (MOM) that is JMS specification compliant.

JMS Client – Concept is similar to MOM clients discussed earlier, the only distinctive factor in JMS world is, these are java based senders and receivers.

JMS Message – Similar to MOM message discussed earlier, JMS message represents data. But JMS brings standardization in terms of message structure and header fields exchanged between clients (sender and receiver).

JMS API – Set of standard interfaces-libraries based on java language which programs can use to connect to JMS Provider and access the facilities.

JMS Administered Objects - Administered objects are JMS objects which are created by an administrator. There are two types of JMS administered objects:

- *Destination* - This object is used by JMS client as the target or source of messages (depending on whether JMS client is sender or receiver of the message). So *Destination* essentially refers to Queue or Topic.
- *ConnectionFactory* - This object is used by JMS client to create a connection with a JMS Provider (Messaging System).

The diagram below depicts all the entities discussed above from JMS perspective.

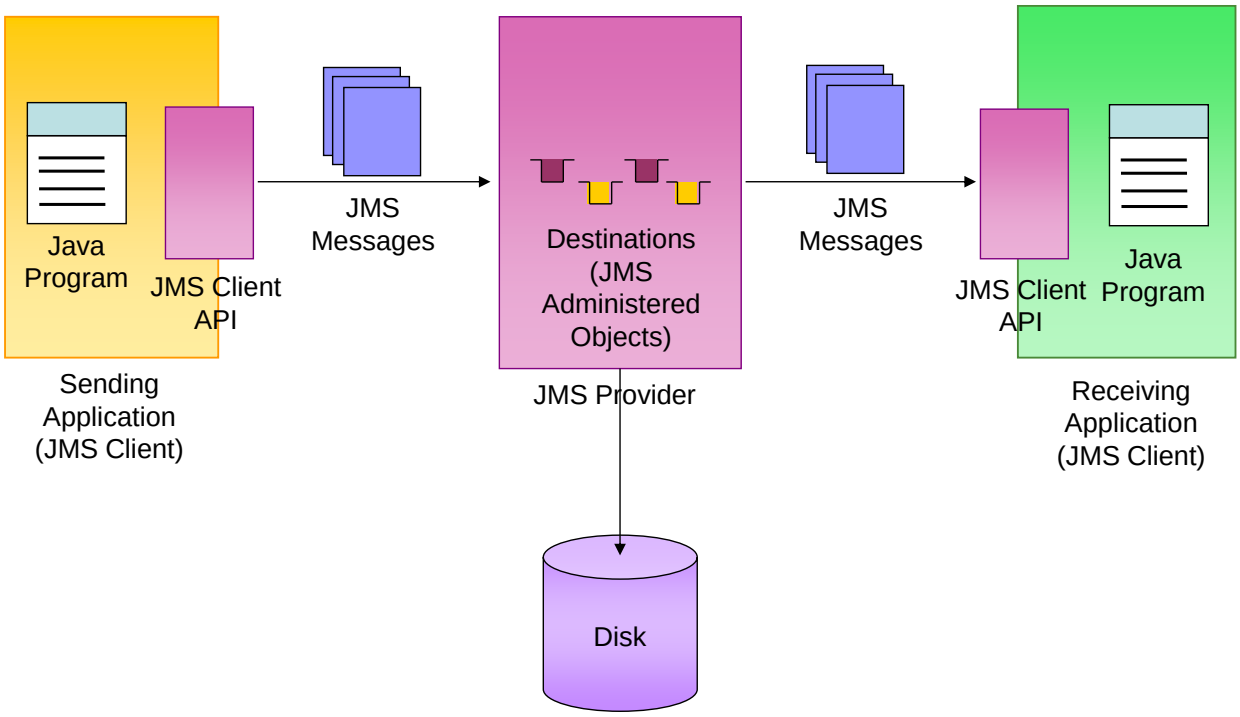

Figure: JMS Core Components

In addition to compliance with JMS specification (ensuring standardization), leading Messaging vendors also provides additional and advanced features.

Going forward we will use Messaging system and JMS Provider interchangeably to refer to MOM.

JMS Messaging Styles

There are 2 types of messaging styles, applications can use one of these 2 styles depending on the requirement for exchanging the data –

- Point-to-Point (PTP)
- Publish-Subscribe (Pub/Sub)

Point-to-Point:
The fundamental of this messaging style is that, at any point in time, the message is delivered only to one consumer; there cannot be two or more consumers of the same message. Point-to-Point style of messaging uses queue as the destination. Message Producer sends the message to the queue and Message Consumer consumes the message from the queue.

Once the message is delivered to the consumer and the Messaging system receives acknowledgment from the consumer, it deletes the message from queue. In case the consumer application is down temporarily, the messages are always retained in the queue until consumer consumes them or messages expire.

Point-to-Point messaging style is used in 2 types of data exchange patterns

- *Request-Reply*
- *Fire-and-Forget*

Request-Reply:
Scenarios where sender of the message expects the response message from the receiver of the request message. Sender application waits for the response. Receiver of the request message responds back to sender after processing the request in real-time.

In request-reply scenario, sender of the request message plays a role of Service Consumer (or Service Requester) and receiver plays a role of Service Provider.

There could be multiple instances/threads of Service Provider application (receiver application) running to provide better load balancing feature by connecting to the same queue. But the fundamental of point-to-point style remains the same - message is delivered to only one instance/thread for processing; other load balanced instances do not receive the same message. Usually distribution of messages coming to queue is done in round-robin fashion across load balanced instances/threads of receiver application.

Now let's take a quick look at an example from the Telecom domain

Scenario 1: Bill Details
Pattern: Request-Reply
Domain: Telecom

The diagram below depicts the scenario where Call Center application retrieves bill details of a particular customer from back-end Billing application while dealing with an on call customer. This scenario is a candidate for point-to-point messaging style – Request-Reply pattern. Call Center application sends Bill Details Request to Billing application via Bill Details Request queue and Billing application responds back with Bill Details via Bill Details Response queue.

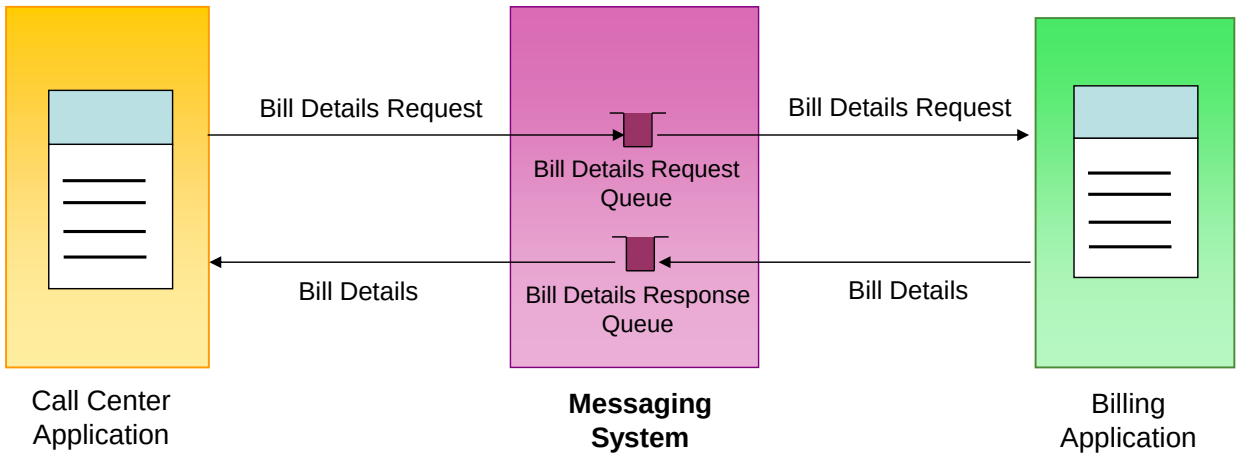

Figure: Point-To-Point Messaging Style: Request-Reply Pattern

Fire-and-Forget:
Scenarios where sender of the message do not expect the response from the receiver of the message. Receiver of the message just consumes the message in real-time.

In Fire-and-Forget scenario, usually message represent some event which has taken place in sender application. So sender plays the role of Event Producer and receiver plays a role of Event Consumer.

Now let's take a quick look at an example from the Banking domain

Scenario: Funds Transfer event
Pattern: Fire-and-Forget
Domain: Banking

The diagram below depicts the scenario where Internet Banking application generates Funds Transfer Event for an online funds transfer initiated by the customer and sends this event to Fraud Management application so that any fraudulent patterns can be detected. This scenario is a candidate for point-to-point messaging style – Fire-and-Forget pattern. Internet Banking application sends Funds Transfer Events to Fraud Management application via Funds Transfer Event queue. Note that the event flows only in one direction from Event Producer to Event Consumer.

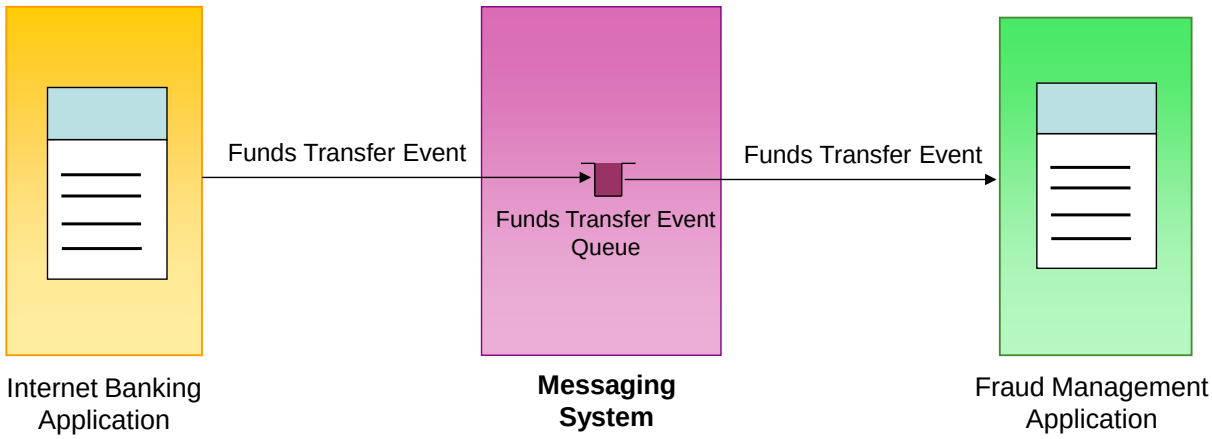

Figure: Point-To-Point Messaging Style: Fire-and-Forget Pattern

Publish-Subscribe:

The fundamental of this messaging style is the distribution or broadcasting of information to multiple interested consumers/subscribers. Information is broadcasted through pre-defined topic (subject). Information Publisher addresses the message on a particular topic (subject) and all the subscribers on that particular topic receives the message.

Publish-Subscribe style of messaging uses topic as the destination for information distribution. There can be more than one publisher of the information and typically there will be multiple subscribers who all will receive the message from the publisher.

In this kind of messaging style, publisher of the message is not interested in receiving any reply/acknowledgement from the subscribers, the only objective is to broadcast the message to all interested listeners or subscribers.

There are 2 types of subscribers –

- Non-durable
- Durable

In case of non-durable subscriber, if the subscriber is not active when the message is published to the topic on Messaging system, that particular subscriber will not receive the message; all other active subscriber receives the message. In order to ensure that the business critical messages are not lost in situation when subscriber is temporarily inactive, subscriber has to register with the Messaging system in durable subscription mode. Messaging server keeps the copy of the message as long as there is any durable subscription on the topic or until message expires. Messaging server delivers copy of retained messages to durable subscriber whenever it becomes active again.

Now let's take a quick look at an example of online Retailer

Scenario: Offer Broadcasting
Pattern: Publish-Subscribe
Domain: Retail – B2C e-commerce

Let's assume e-Retailer ABC has multiple channels like Call Center, Retail Outlets, Online Portal, Partner Portal and so on to sell electronic products. As per company strategy, during festive season lot of time-bound promotional offers are given to the customers on various electronic products. These offers need to be consistently available across all the channels in a timely manner. This is the classic case of Publish-Subscribe messaging style. It doesn't make sense for various channel applications (managing customer interactions) to keep requesting offers from Offer Management system. However, such scenario can be more effectively addressed if Offer Management application keeps publishing new offers and expired offers. Interested channel applications can subscribe to these offers and update offers in their respective systems. In this case Offer Management application will not require response from any of the subscriber channel applications. Now if any new application is also interested in offers, it can be added dynamically, by subscribing to the same topic 'Offer' without any changes in Offer Management application. The Publish-Subscribe scenario is depicted in the diagram below; as shown the copy of message is delivered to each subscriber.

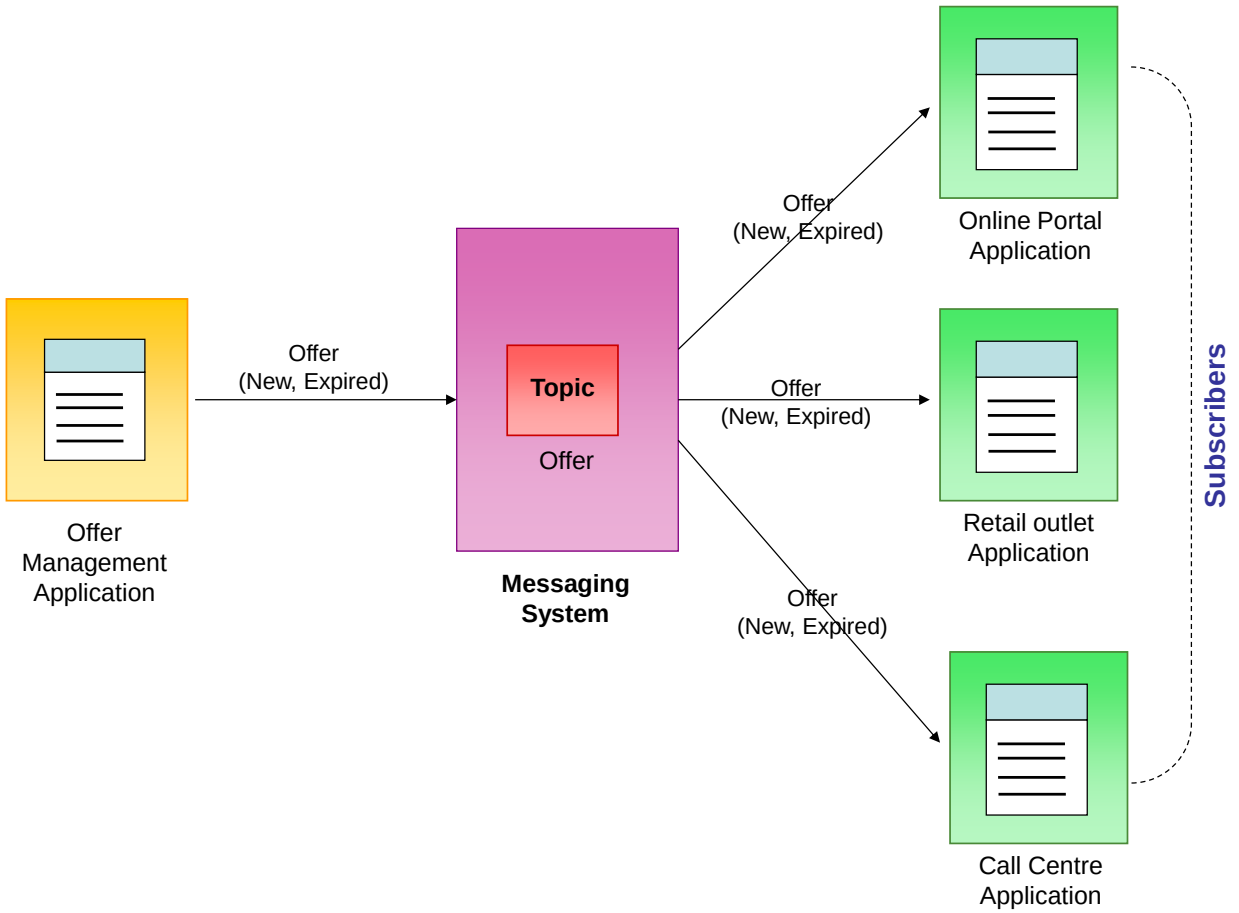

Figure: Publish-Subscribe Messaging Style: Information Distribution

So in a nut shell, Messaging system can be used by Enterprise applications as integration layer for exchanging data in various kind of scenarios – Request-Reply, Fire-and-Forget and Information distribution.

Message Structure

Basically messages are data that the sender application sends to receiving application. The content of the message could be any acceptable format as agreed by source and target system.

JMS specification defines the standard structure for the message. The structure is composed of the following parts:

- Header
- Properties
- Body

Header – The Header segment contains standard pre-defined properties for the message.

Properties – Properties segment provides facility for adding optional header fields in the message. Properties segment could contain application specific optional header fields or JMS Provider specific fields.

Body – This segment is the segment which actually carries the business data/content. JMS defines different types of message bodies ensuring all types of messages can be exchanged Ex. Text, Binary

Only Header is mandatory, Properties and Body are optional

Let's have a look at Header, Body and Properties in detail

JMS Header:

JMSDestination:
This header field contains the name of the destination to which the message is being sent. Sender sets this field to particular queue or topic destination.

Set By: *Send* method

JMSDeliveryMode:
JMS supports 2 types of delivery modes

- PERSISTENT
- NON-PERSISTENT

PERSISTENT – When PERSISTENT mode is set on the message, it instructs JMS Provider to persist the message in storage, typically either in file store or DB store whichever is configured.

NON-PERSISTENT – When NON-PERSISTENT mode is set on the message, it instructs JMS Provider not to persist the message in storage, NON-PERSISTENT messages are kept in memory.

Set By: *Send* method

Significance of PERSISTENT and NON-PERSISTENT message will be discussed in *'Avoiding Data Loss'* section.

JMSMessageID:
This field uniquely identifies each message sent by a Message Producer

Set By: *Send* method

JMSExpiration:
This field specifies the duration of time the message should remain in the destination (queue or topic) before expiration. If the message is not consumed within the time duration set in *JMSExpiration*, it becomes unavailable to receiver/consumer after that. Handling of the expired messages depends on the mechanism implemented by individual JMS Provider. JMS Provider may choose to destroy the expired message or may provide the facility to route the message to other destination upon expiration, where a separate processing can be associated to handle these expired messages.

The simple example of this could be the Bill Details request message discussed in point-to-point messaging style. The Call Center application will place the Bill Details request message to request queue with certain expiry which is less than or equal to time the Call Center application will wait before timing out in case of no response from Billing application. Another example of message expiry could be short lived-offer generated by Offer Management application for channels, if these time-bound Offers are not consumed by channel applications in timely fashion, then they are of no meaning and should expire. In both the above situation, *JMSExpiration* feature is useful making messages to live only for certain duration.

Zero in *JMSExpiration* indicates no expiry. But again the value may be different for different Messaging vendors.

Set By: *Send* method

JMSReplyTo:
This field is used in Request-Reply scenario to specify destination on which the reply to the request message should be received.

Sender application sets this header field before sending the request message to the Messaging system. Once receiver application (Service Provider application) processes the request message, it must send the reply back to the destination specified in *JMSReplyTo* field.

This field is left blank in case of Fire-and-Forget scenarios where no response is expected from the consumer of event.

There may be multiple applications requesting the same service from the Service Provider application. The diagram below depicts the scenario from Banking domain where same Account Summary service is shared across multiple channel applications.

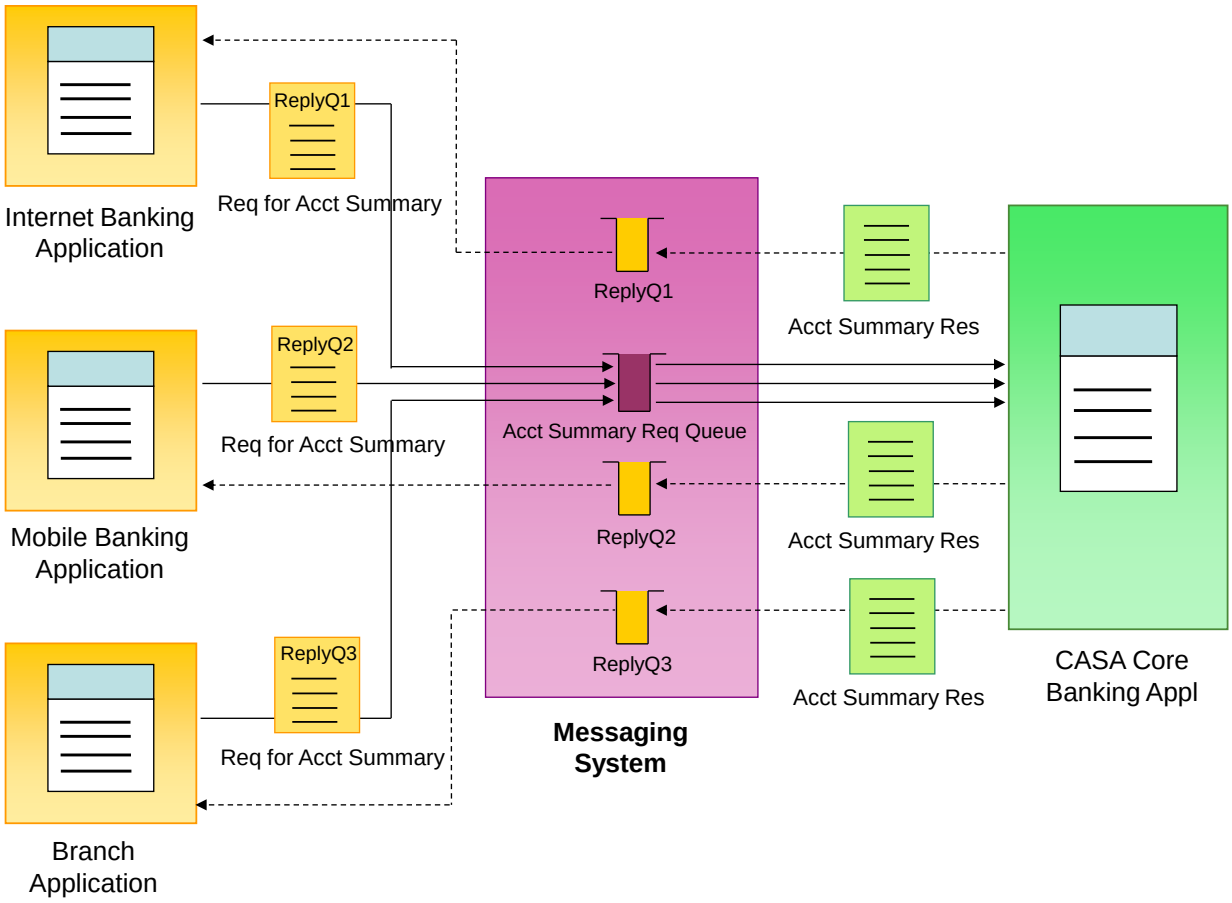

Figure: ReplyTo Queue scenario

In this case, Account Summary is retrieved from CASA (CASA – Current Account, Savings Account) core banking application after customer logs in successfully via Internet Banking/Mobile Banking or visits the Bank branch. While multiple transactions are going on simultaneously for different customers through these multiple customer facing channel applications, data is requested from the same back-end provider (CASA core banking application). It is important that response/reply data goes back to appropriate front-end channel application. CASA application being agnostic about which channel application is requesting the details, relies on the *JMSReplyTo* header field to send back the reply. So it is requesting application's responsibility to set the appropriate queue name in *JMSReplyTo* field while sending request message to Service Provider application; requesting application then listens on this queue for reply.

Set By: Client

JMSCorrelationID:
This field is used in Request-Reply scenario to correlate the response message back to the request message sent by the application.

Sender application sets this header field before sending the request message to the Messaging system. *Correlation ID* can be any unique string or number. Once receiver (Service Provider application) processes the request message, it just copies the correlation ID from request header back into response header *JMSCorrelationID* field.

This field is left blank in case of Fire-and-Forget scenarios where no response is expected from the consumer of event.

The diagram below depicts the scenario of Call Center application sending multiple requests to back-end Service Provider Billing application to get Bill Details (Multiple requests by same Call Center application is due to customer service agents attending multiple customers simultaneously, which may be the typical case).

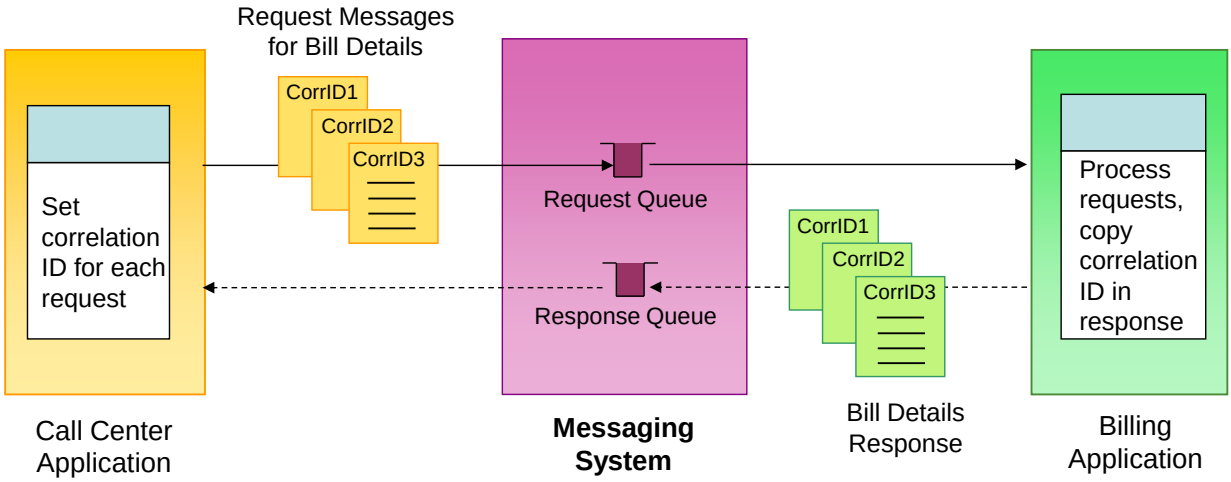

Figure: Correlation ID scenario

While ReplyTo queue for getting customer Bill Details for all requests may be same, it is important for Call Center application to link right response message back to each individual customer Bill request it has sent. Billing application being agnostic about this, it just copies the correlation ID from request header back into response header, *JMSCorrelationID* field. So it is requesting applications responsibility to set the unique correlation ID for each request it is sending so that when the reply comes back from Service Provider application, it can link it to the corresponding request.

Set By: Client

JMSRedelivered:
This field is applicable to only Message Consumer and it is set by Messaging system. When this field is set to true, it indicates that message is being redelivered to consumer by Messaging system. This situation occurs when consumer fails to send acknowledgement of received message to Messaging system; Messaging system assumes that message delivery

to consumer has failed as there is no acknowledgement so it redelivers message to consumer and sets this field as true to indicate it is redelivery.

Set By: Messaging system

JMSType:
This field contains type of the message. Some Messaging system may have repository that contains the schema or definition of messages flowing through it. In such cases, the Message Producer may populate this field with reference to definition of message it is sending.

Set By: Client

JMSPriority:
This field contains the delivery priority as defined by JMS specification. This field is set by Message Producer, priority range between 0 to 9 with 0 being the lowest priority and 9 being the highest. For example, a Banking channel application sending payment transactions to back-end payment engine may set a high priority for high value payment transactions as compared to low value payment transactions.

When the priority is set to higher value, Messaging system is expected to deliver high priority messages to consumer before delivering messages with lower priority. This behavior is applicable only when there is backlog of messages in the queue. If there is no backlog of the messages and there is only one message to deliver, priority has no meaning.

Set By: *Send* method

JMSTimestamp:
This field contains the time the message is sent to the Messaging system.

Set By: *Send* method

JMS Properties:

Property segment provides the facility for adding optional header fields in the message. Properties are defined as key-value pair. Property names must follow the rules as identified by JMS.

Properties segment can contain

- JMS Defined properties
- JMS Provider specific properties
- Application specific properties

JMS defined properties:
JMS reserves property name prefix "JMSX" for JMS defined properties
Example:

JMSXAppID – Identifies the application sending the message
JMSXUserID - Identifies the user sending the message
JMSXGroupID – Identifies the group to which this message belongs

JMS defines one mandatory property – *JMSXDeliveryCount*. The Messaging system sets this count whenever it delivers the message to consumer. It represents the number of times message has been delivered. For the first time, it is set to 1. If the message is being redelivered sighting no acknowledgment from consumer, then value of this field will be 2 or more, at the same time *JMSRedelivery* field will also be set to true. Consumer can ignore redelivered message, if it has received and processed the first message successfully, but somehow failed to acknowledge the receipt of it Messaging system. So these indicators helps consumer to avoid duplicate processing.

JMS Provider specific properties (Messaging system specific):
JMS reserves property name prefix "JMS_*vendorname*". Messaging system vendors are free to define their own set of property names. The primary purpose of giving the provision is that the Messaging vendors can offer more features/facilities which may not be part of JMS specification. The example of one such feature is message compression. Compression will facilitate efficient transport and storage of messages in Messaging system.

Application specific properties:
JMS does not reserve any specific name prefix for application specific properties; any name that does not start with "JMS" or "JMSX" can be used to specify application property. The application specific properties are primarily used in Message Selection. We will see use of application specific property in '*Message Selector*' section.

Property values can be of type boolean, byte, short, int, long, float, double, and String.

JMS Body:

This segment of the message contains actual data. The message can contain one of the following message bodies

- TextMessage
- MapMessage
- StreamMessage
- ObjectMessage
- BytesMessage

TextMessage:

Represents message body that contains String. XML being the most common way of exchanging data between applications, Text messages will be used most with XML data as content.

MapMessage:
Represents message body that contains set of name-value pairs, where names are java String objects and values are java primitive types. The entries can be accessed sequentially or directly by name.

StreamMessage:
Represents message body that contains stream of java primitive values. Values must be read sequentially.

ObjectMessage:
Represents message body that contains serializable java object.

BytesMessage:
Represents message body that contains stream of bytes.

Now after understanding message structure (Header, Properties and Body), let's understand destination in detail.

Destination

As discussed in the earlier chapter, there are 2 types of destinations –

- Queue
- Topic

Queue is used as a destination in Point-to-Point messaging style, while Topic is used as a destination in Publish-Subscribe messaging style.

Common practice is to create dedicated destination for each individual business transaction or event. For example, a dedicated queue *Account Summary* request, a dedicated queue for *Bill Details*, dedicated Topic for *Offer* event.

Queues and Topics are usually created upfront using administration capabilities, these pre-defined destinations are referred as static destinations.

In addition to static destination, there is a notion of temporary destination (temporary queue and temporary topic). Temporary destinations can be created by client applications on demand at runtime. The scope of temporary destination is that of the connection or session that is used to create temporary destination by client. Temporary destination is a unique object created and have a specific purpose; they are typically used as a destination

for reply messages. Sender application can create temporary destination on the fly and set it in *JMSReplyTo* field of request message, this is to ensure that reply to each request is received on unique and individual temporary queue. This approach is primarily used in multi-threaded scenario, where each thread of a particular application will be sending request to the same static request queue but will be waiting for response on individual temporary queue. Request sending application must delete temporary queue once it receives the response message or it is deleted from messaging server once the connection or session that created it, is closed.

The diagram below depicts the scenario related to temporary queue for Customer Details request-reply flow. As shown in the diagram, each listener thread will listen on its own unique temporary queue for reply.

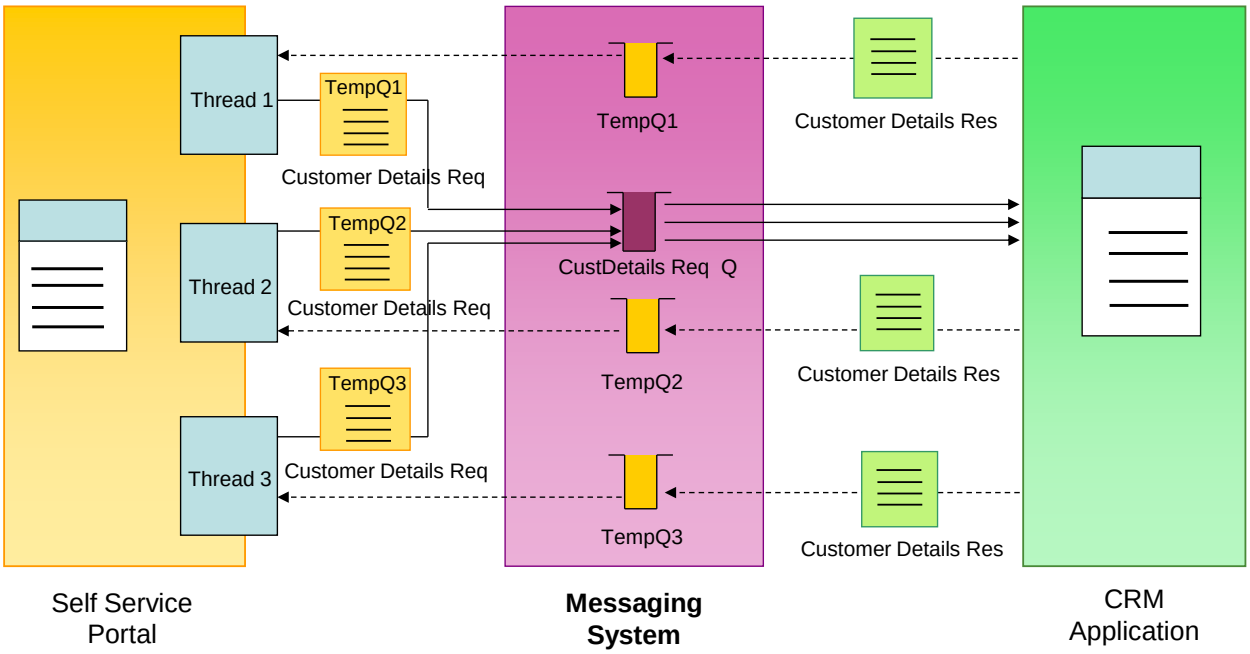

Figure: Temporary destination scenario

If temporary queues are not used for replies, then there is always an option to use single pre-defined static queue to receive replies; in such cases all listener threads are linked with single response queue and individual thread need be apply some logic to select the appropriate response, usually based on JMScorrelationID.

So in summary, static destinations are managed by administration facilities provided by Messaging vendor. Administration allows for creation and deletion of static destinations. On the other hand, temporary destination has to be explicitly created by client and it gets deleted when the connection or session is closed. Use of temporary destinations for reply messages is the preferred mechanism due to flexibility and scalability it offers.

JMS doesn't specify any guidelines or restriction on naming conventions for static destination. It is up to Organizations to define naming convention for static destinations.

Destination Properties:

Messaging system defines properties on destination. Some of them could be specific to type of destination - Queue or Topic. The primary purpose of properties is to provide control over specific behavior. Some of the common properties available with Messaging systems are –

Destination Capacity:
The purpose of this property is to define capacity for a particular destination. Capacity of a destination has two aspects

- Maximum number of messages: Maximum number of messages a destination can hold.
- Maximum size: Total size, combining sizes of all messages that a destination can hold.

Capacity is determined using following key factors

- Total number of transactions per hour (average volume and peak hour volume)
- Size of each message
- Maximum recovery time of consumer application, in case of unforeseen failure

Taking an example of Bill Details interface between Call Center to Billing application, let's assume below are the numbers

- Total number of transactions per hour from Call Center to Billing application: average volume - 1500 per hour and peak volume – 2000 per hour
- Average size of each message: 1 kb
- Maximum recovery time of consumer application: 2 hours

Based on above numbers, capacity of Bill Details Request queue can be configured as below

- Number capacity: 4000 messages
- Size capacity: 4 MB

If the capacity is exhausted to the limit, any new incoming message from Message Producer may be rejected by Messaging system; this behavior is all Messaging system specific.

Message Ordering:

Sometimes ordering of message delivery from destination is very important. Having message priorities can impact this ordering behavior. This necessitates **Message Order** to be enforced at destination level. This will ensure that messages are delivered in first in, first out order without honoring the priority set in individual incoming message.

Producer Flow Control:
Sometimes consumer falls behind in processing the messages due to some or the other reason. In such situations, it becomes important to control the flow of incoming messages to the destination to avoid memory or capacity being exhausted for that particular destination. So essentially, this is a scenario where consumer becomes slow in processing messages temporarily, while producer is producing messages at a faster rate.

To control this behavior, Messaging system usually provides property to control the incoming message flow on destination. Messaging system allows to configure threshold on destination capacity, whenever it detects that the threshold of unconsumed pending messages has reached (due to slow consumption of messages by consumers), Messaging system slows down the producer by using one of the following mechanism

- Block the send call until some messages are consumed and some capacity becomes available for next messages. In this case producer simply waits for call to return from Messaging server.
- Choose to throw exception to producer, in which case producer can wait for some time and then retry sending the message.

Avoiding Data Loss

Avoiding loss of business critical transactions being exchanged between applications is one of the most critical requirements in any Organization. Avoiding data loss is specifically applicable in Fire-and-Forget scenarios where it is one way flow of data from sender to receiver and sender does not expect any response back. So it becomes very important to ensure message is delivered to the recipient in a guaranteed manner.

In this section, we will see what all it takes to accomplish the successful message delivery and processing. Following are the 3 important aspects to avoid data loss

- Message Delivery Mode
- Message Expiry
- Message Acknowledgement

Appropriate configuration of these 3 aspects will provide utmost assurance that message is successfully delivered to consumer and processed successfully.

Let’s take a look at these aspects one by one

Message Delivery Mode: PERSISTENT and NON-PERSISTENT

PERSISTENT – PERSISTENT messages are persisted either in file store or DB store whichever store is configured. So even if JMS Provider fails/restarts before delivering the message to the consumer, there is no loss of data and message is delivered to the consumer once JMS Provider is up and running. This has a bit of performance impact as IO operation is involved. However, it is more reliable as message gets delivered in a guaranteed manner and sustains JMS Provider failure. Upon re-start JMS Provider recovers stored message from storage and delivers it to the consumer.

Typically, PERSISTENT messages are sent in sync mode by Message Producer to JMS Provider, in this case JMS Provider writes the message to store first and then acknowledges or confirms the receipt of message to Message Producer. This ensures PERSISTENT message is safely persisted in the store and can sustain JMS Provider failure. Taking this synchronous and sequential approach of store and acknowledge also ensures that in case of any error while persisting message to store, producer get an opportunity to re-submit the message to JMS Provider.

The diagram below depicts the behavior.

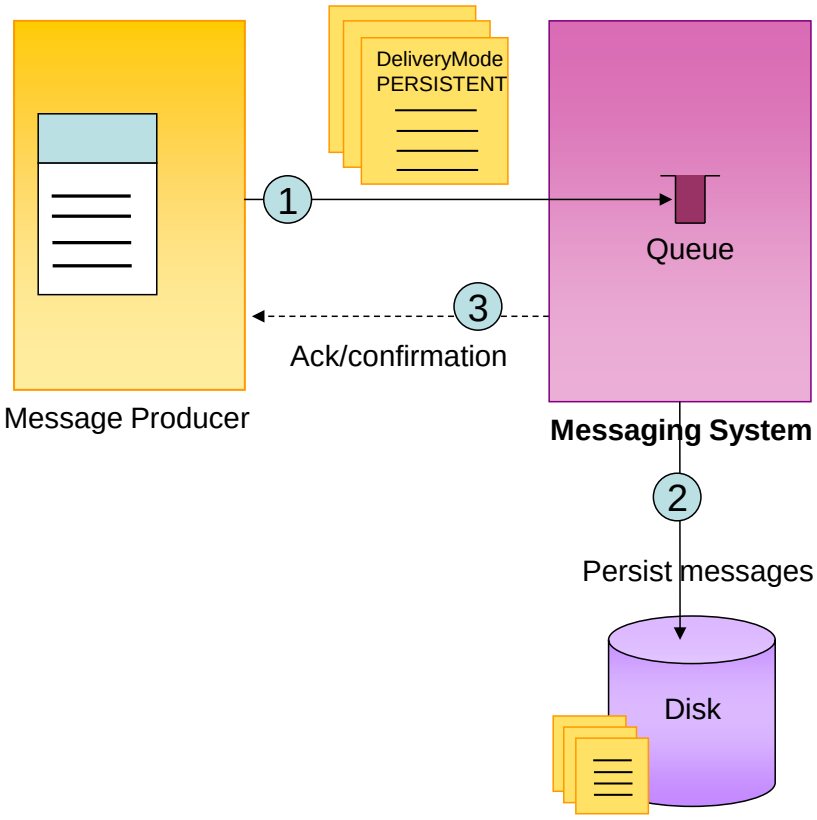

Figure: PERSISTENT messages in sync mode

For some use cases, if application is tolerant to small amount of data loss in case of JMS Provider failure while persisting message to the store, async mode can be used while sending PERSISTENT messages. In this case, writing message to store and sending acknowledgement to producer are done in parallel.

NON-PERSISTENT – NON-PERSISTENT messages are stored in memory. So if JMS Provider fails/restarts before delivering message to the consumer, there is a loss of data as memory is erased. Though this is a better option in terms of performance as no IO operation is involved, it is not as reliable as PERSISTENT messages.

Choice really depends on the trade-off between performance and guaranteed delivery. It is also important to consider the Business Critical nature of the transaction.

For example, if it is a Payment transaction where data loss is not at all acceptable, then sender must set delivery mode as PERSISTENT ensuring guaranteed once-and-only-once delivery, guaranteed delivery takes precedence over performance. If it is simple notification or acknowledgement message, sender may prefer to set delivery mode as NON-PERSISTENT.

Message Expiry:

While *Message Delivery mode* is the most important aspect in avoiding data loss. It is also important set the Expiry of message appropriately considering the possible downtime and recovery duration of consumer application. For example, if recovery mandate for consumer application is 4 hours. (i.e. if it goes down due to unforeseen event, it should recover in max 4 hours), then message expiry should be set to more than 4 hours. If this is not set properly, there is a possibility that message will expire (even though it is PERSISTENT), if recipient application is down for longer duration.

Zero in *JMSExpiration* indicates no expiry. As a common practice, Expiry for Fire-and-Forget messages are usually set to zero.

Message Acknowledgement:

The way message acknowledgements are handled between JMS Provider and Message Consumer also plays an important role in ensuring critical Business events are not lost. With that key objective, it is important that acknowledgement mode between JMS Provider and consumer is configured appropriately.

JMS specification provides with multiple message acknowledgment options; this instructs the JMS Provider on *message retention*, *redelivery* or *deletion*.

Let's take a look at this in detail.

Let's see acknowledgment options in the context of JMS Provider and Message Consumer. The diagram below shows the acknowledgement from consumer to JMS Provider and what it means to JMS Provider (step 4 and step 5). After receiving acknowledgement, the JMS Provider is expected to discard the message from destination and not supposed to deliver it to consumer again.

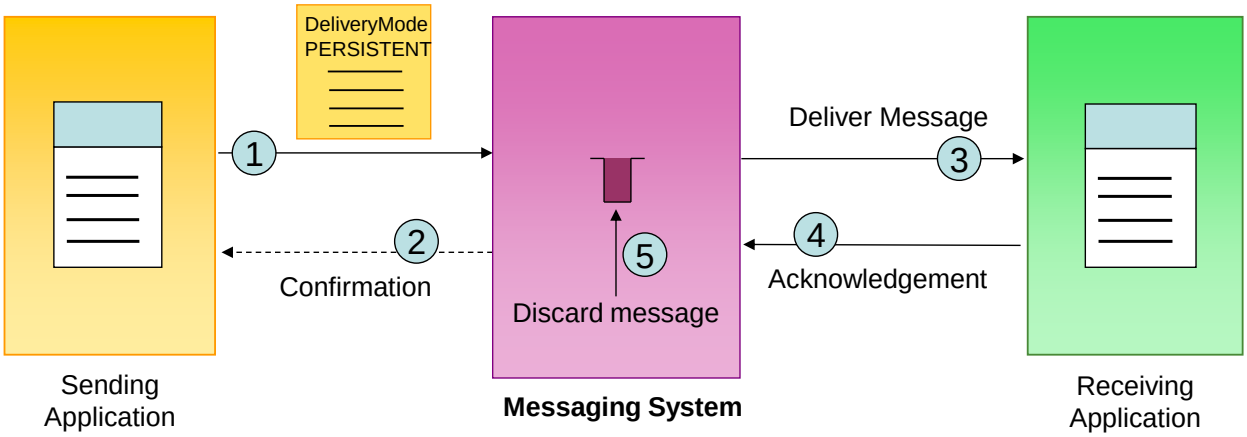

Figure: Acknowledgement

Acknowledgment is handled automatically for transacted session through commit and recovery is handled by rollback. As per JMS specification there are three options available for Message acknowledgment for non-transacted session as mentioned below –

- AUTO_ ACKNOWLEDGE
- CLIENT_ACKNOWLEDGE
- DUPS_OK_ACKNOWLEDGE

AUTO_ACKNOWLEDGE:
When this option is configured, session automatically acknowledges the receipt of the message to JMS Provider. When using this mode consumer doesn't have control over the acknowledgements and it is quite possible that the acknowledgment to JMS Provider is successful but message consumer has failed processing the message successfully. JMS Provider deletes the message as it has received the acknowledgment. But in reality processing of message might fail at consumer end and this will result in loss of data/message.

To ensure that the message gets deleted by server only after successful processing by consumer, CLIENT_ACKNOWLEDGE mode should be used.

CLIENT_ACKNOWLEDGE:
When this option is configured, a consumer acknowledges the receipt of the message to JMS Provider explicitly (by calling message's *acknowledge* method). When using this mode

consumer has direct control over the acknowledgement. Consumer can acknowledge message to JMS Provider only after ensuring that it has processed the message successfully. For example, processing might involve storing the message to database for further use. Use of this acknowledgement mode prevents loss of data/message. In case of processing failure at consumer end, consumer will not send acknowledgment to JMS Provider. JMS Provider will redeliver the message, since it has not received the acknowledgment from the consumer.

DUPS_OK_ACKNOWLEDGE:
When this option is configured, it instructs the session to lazily acknowledge the receipt of message to server, this might result in JMS Provider redelivering the messages to consumer. If this mode is configured consumer is expected to be ok with handling of duplicate messages.

So in a nut shell, it is very important to use *Message Delivery Mode* as PERSISTENT, *acknowledgement mode* as CLIENT-ACKNOWLEDGE and *Expiry* as ZERO to avoid loss of critical business events.

Scenario: Avoiding Data Loss
Let's assume a scenario where Dealer/Retailer is booking the Order through manufacturer's Self Service Portal. The request needs to be submitted to Order Management application for processing. While Portal application immediately provides an acknowledgement of Order Booking to online Dealer/Retailer, it needs to ensure that the Purchase Order event successfully reaches the Order Management application. If the message is lost in the process of being transferred to Order Management application and Order fulfillment does not take place, it will lead to dissatisfied Dealer/Retailer. For manufacturer, this would result in unhappy customer and even loss of future Business.

So as discussed in earlier sections, for maximum reliability the overall design consideration should be something like this –

- Self Service Portal should send Purchase Order message with *message delivery mode* as PERSISTENT and with *Expiry* set to zero.
- Order Management application should use CLIENT_ACKNOWLEDGE mode and acknowledge the receipt to JMS Provider explicitly only after storing Order data to database for further processing.

The diagram below depicts the overall scenario.

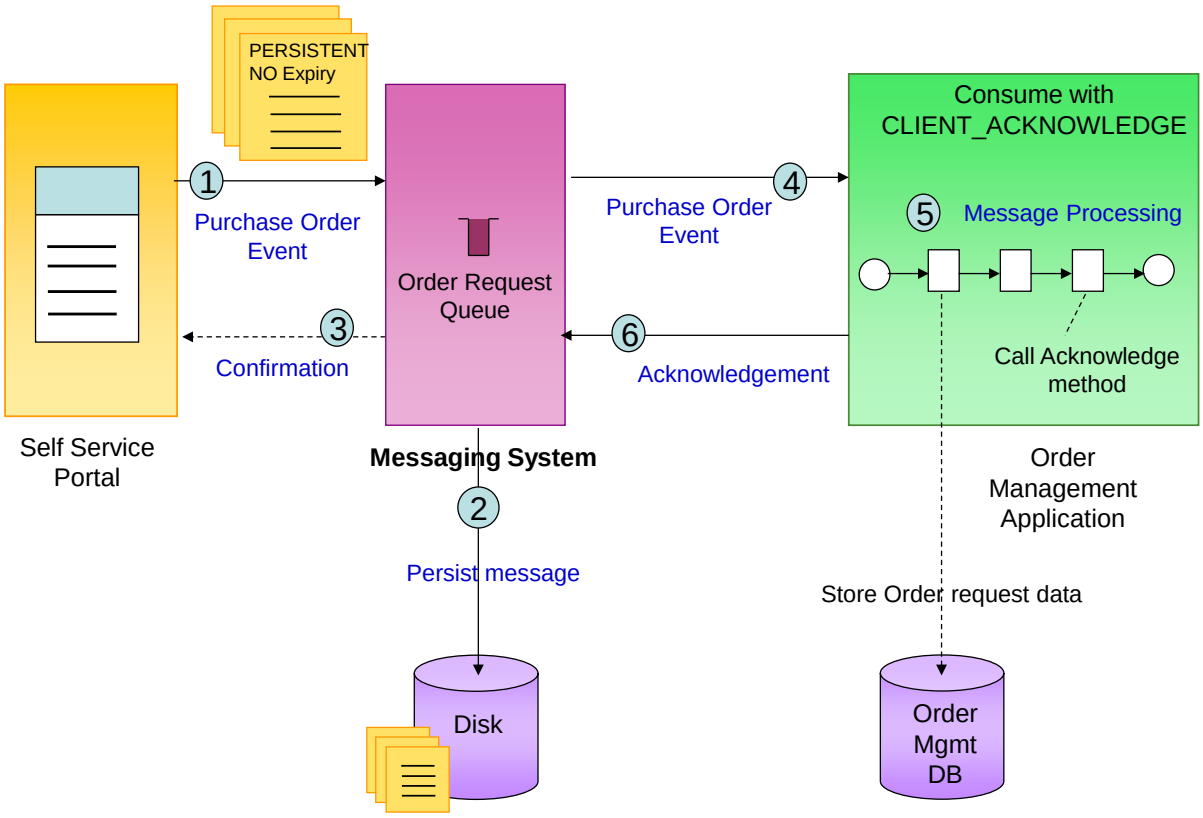

Figure: Avoiding Data Loss – Purchase Order Event

Purchase Order event is just one example, but this is applicable to any business critical transaction. For example, customer initiating funds transfer using internet banking channel; in this case the payment instruction goes from internet banking channel application to payment engine and data loss is not at all acceptable situation.

Message Selector

Message selectors allows a consumer connected to a particular destination to select the messages it is interested in. Selector applies only to the values present in message header and message properties and cannot reference values present in message body, this is primarily because header and properties are key-value pair while body could be any structure – fixed width, delimited, XML etc.

Application specific message properties are primarily used for message selection purpose. Based on the requirement, Message Producer sets the transaction specific metadata in JMS properties and consumer can select the message it is interested in by specifying appropriate selection criteria. Selection criteria allows message consumer to skip the unwanted messages.

In a Point-To-Point domain, when a message consumer connected to a queue uses *QueueReceiver* interface with a message selector, it receives the messages that matches the criteria; unselected messages remain on the queue and are available to other consumers connected to the same queue.

In a Publish-Subscribe domain, when a message subscriber connected to a topic uses *TopicSubscriber* interface with a message selector, it receives the messages that matches the criteria; unselected messages just do not exist from subscriber perspective.

A message selector is nothing but a string whose syntax is based on SQL 92. When the selector evaluates to true it matches the message. Selector expression comprises of identifiers and operators – logical, comparison and so on. Ex. "AmountOfTransfer>= 1000".

Let's look at the case study below from telecom operator domain to understand need of selector.

Scenario: Message Selector

When customer logs in through Telecom operator's Self-Service Portal, depending on customer's profile whether the customer is individual customer or corporate customer, appropriate account information needs to be fetched and displayed to the customer. Typically, Telecom Organization will have separate Billing systems for managing accounts of individual and corporate customers. The Retrieve Account Details request gets submitted from Self-Service Portal application to the Messaging system and should reach appropriate Billing application (Retail Billing application or Corporate Billing application) based on *Customer Category* to fetch the account details.

The Portal application sets application specific property called CustomerCategory=*Category* in JMS properties of Retrieve Account Details request message before submitting it to the Messaging system.

Portal application side, the code might look like as below –

```
accountDetailsRequestMessage.setStringProperty ("CustomerCategory", Category);
```

At runtime, *Category* value can be 'Individual' or 'Corporate' based on customer's profile.

Retail Billing application and Corporate Billing application need to specify appropriate selector while selecting the messages from queue as shown in the diagram below.

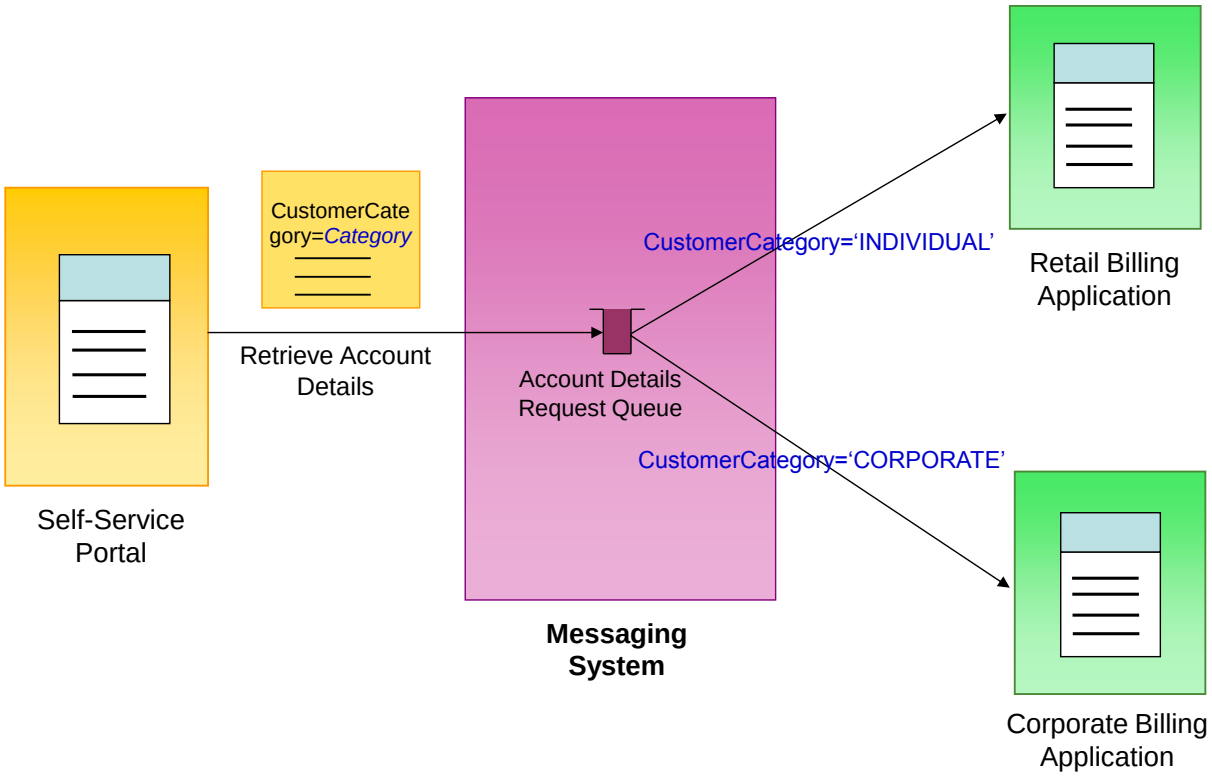

Figure: Message Selector Scenario

Message Compression

Exchanging compressed messages is one of the important aspects when using Messaging system; in certain scenarios, it is beneficial than exchanging uncompressed messages.

Sending application can compress the body of the message before sending it to the Messaging system and receiving application can uncompress the body after receiving the message from the Messaging system.

Compressing message makes sure that it takes less memory space or disk space on Messaging system. This feature is particularly useful for messages of larger size with PERSISTENT delivery mode as it not only takes less space on disk but also ensures messages are handled faster by the Messaging system.

However, at client side (sending and receiving application side), it may have little performance impact while dealing with the compressed messages (due to additional compression/decompression logic). So it is important to consider the overall performance and response time before choosing compression option.

Message Consumption Mechanism

A Message Consumer can consume message from destination in 2 ways – synchronous or asynchronous.

Synchronous:
A consumer can request message from the Messaging system one by one. In this case, consumer keeps polling the Messaging system at regular interval for the next message or waits for the next message.

Asynchronous:
In this case consumer registers a listener object with the Messaging system. In case of JMS, this listener object must implement *MessageListener* interface. Whenever message arrives at destination for consumer, Messaging system delivers it to the consumer by calling listener's *onMessage* method (in case of JMS). While receiving message asynchronously consumer is not blocked and it can perform other activities.

Exclusive Consumer

Typically, queue is FIFO behavior, first in, first out. The first message which comes in queue (from producer), goes out of the queue first (to consumer). So this will maintain the order of the messages received on a particular destination. However, if there are multiple consumers or sessions consuming the messages from the same destination concurrently in a multi-threaded environment, then processing of messages in order is not guaranteed, even though the messages are delivered to these threads in order. A particular consumer thread may process message earlier than other thread and this can disturb the order of processing. In some use cases it is very important to maintain the order of the message processing.

Taking Telecom operator scenario – some offers are valid only if customer subscribes for a particular plan. For example, unlimited download offer applicable only if customer subscribes for $50 or above plan. So essentially, back-end application need to process both the messages in order as well as immediately, so that customer can start availing unlimited download feature as soon as he or she subscribe for the $50 plan, thus enhancing customer experience. While *plan* and *offer* messages are getting submitted one after other immediately, it should not happen that *offer* message gets processed earlier than *subscription plan* message, the *offer* might get rejected (this might happen if there are multiple consumers on destination and messages are getting processed concurrently by message consumer threads).

In such cases, the JMS Provider usually provides with the feature by which destination can be marked as exclusive, this is called exclusive consumer (or exclusive queue); messaging server will pick up only one consumer to deliver all the messages, thus ensuring order of processing. Other consumers on the same destination will be in standby mode. If primary

consumer fails, then only JMS Provider will deliver messages to the next available consumer. Exclusive option is applicable only to queues.

Apart from supporting the core messaging fundamentals discussed so far, the full-fledged messaging product also cover aspects related to

- Administration
- Security
- Fault Tolerance
- Advisory Messages and Monitoring

It is up to JMS Provider how they implement these features in their system. Let's look at the common fundamentals around *Administration* and *Security*. We will look at the other 2 aspects *Fault Tolerance* and *Advisory Messages and Monitoring* in detail in '*Messaging System Advanced Concepts*' Chapter.

Administration

Administration is an integral part of any Messaging system. Typically, Messaging systems provide GUI-console based option to perform various administrative functions. Some vendors do provide administration APIs which allows for creating custom application to manage messaging environment. Some of the common administrative activities performed by administrator are

- Create-Delete-Update-Browse destinations, configure destination properties
- Security configuration – Create-Delete-Update-Browse users-groups, grant or revoke send-receive permissions on destinations
- Create-Delete-Update-Browse Server-to-Server connection
- Purge destinations (clear messages) - queues or topics
- Monitor statistics of destinations – track how many messages are sent and consumed from destination, how many consumers are connected, number of un-delivered messages, and so on
- Start or Stop Messaging server

Security

Two fundamental aspects of security from messaging perspective are

- Authentication
- Authorization

Authentication:
Each client connecting to the Messaging system needs to be authenticated as a valid client. Client can belong to particular group or may have individual identity. So essentially, Message Producer and Message Consumer applications will be authenticated before they can send or consume messages respectively; SSL certificate based mutual authentication is a common practice.

Authorization:
Once client is authenticated, Messaging system will authorize client to ensure it has appropriate permission to send or receive the message from a particular destination. Client may have permission to send messages to particular destination but may not have permission to send messages to other destinations, so permissions needs to be configured accordingly at destination level.

Messaging Platform Component Topology

The diagram below depicts *Messaging Platform* component topology and shows models and key features supported by Messaging products to address messaging based integration requirements.

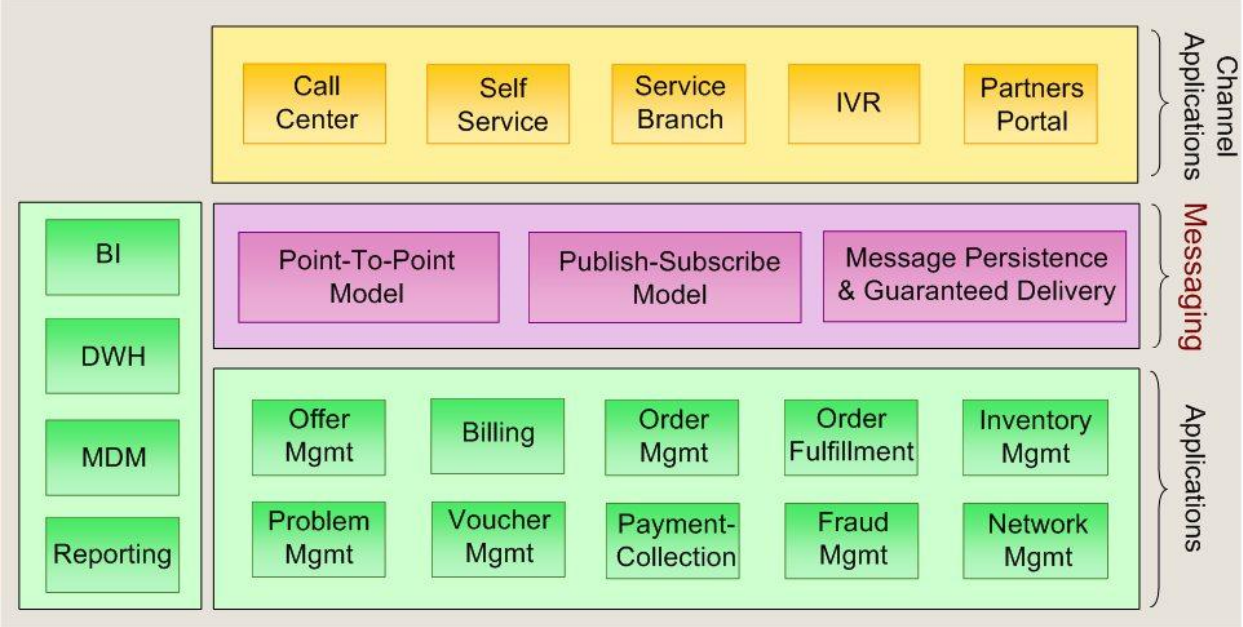

Figure: Messaging Platform Component Topology

Enterprise Real-time Integration Reference Architecture: Messaging

The diagram below depicts the simplistic view of Enterprise Real-time Integration Reference Architecture: Messaging with Banking domain applications as an example.

Though it is shown from channels perspective, it is applicable in general for integrating any Enterprise applications using Messaging system. It depicts how applications can be loosely coupled with each other through messaging layer and how messaging layer supports various integration scenarios between applications: Request-Reply, Fire-and-Forget events and information publication through destinations (*Queues* and *Topics*).

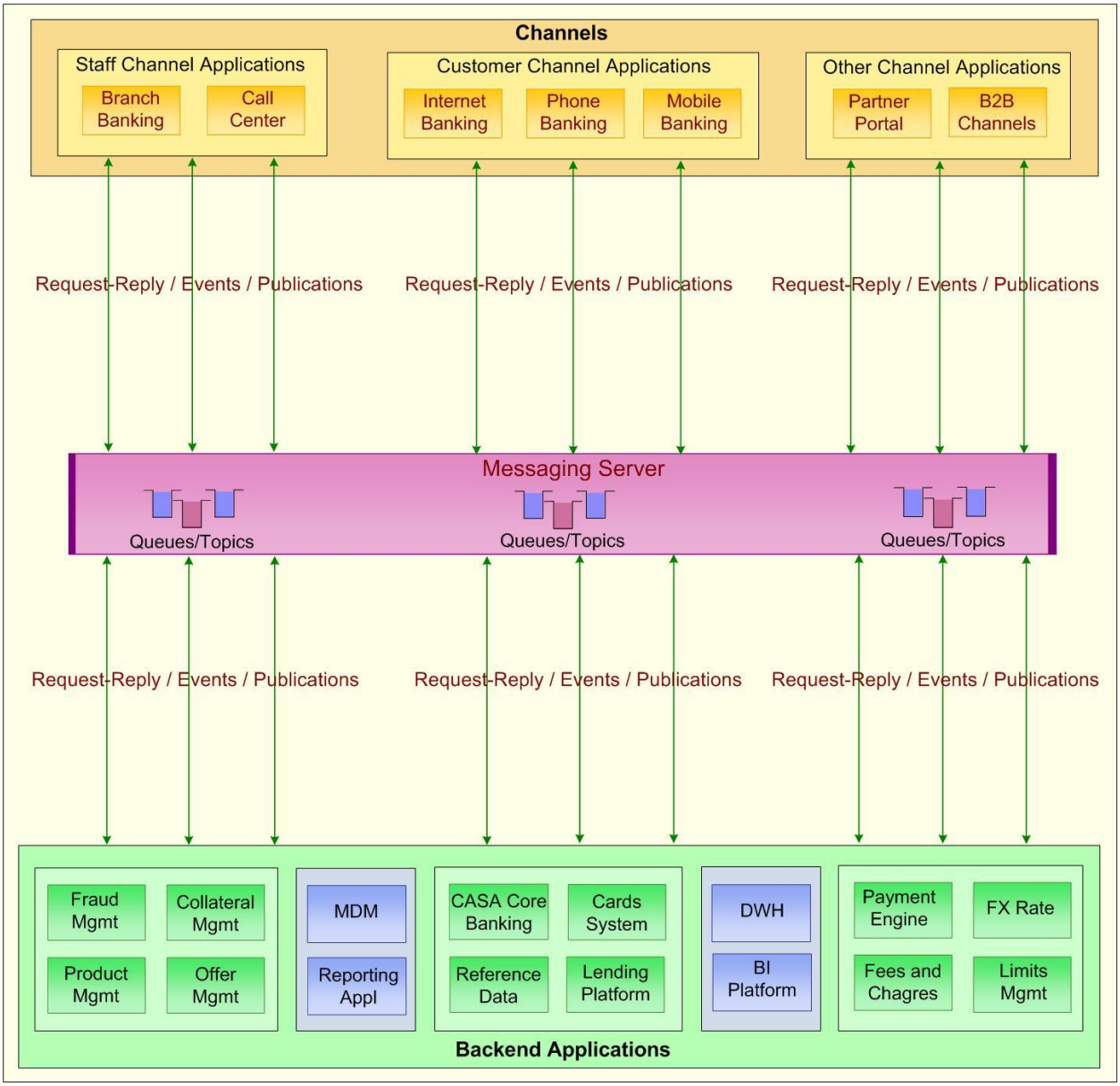

Figure: Enterprise Real-time Integration Reference Architecture: Messaging

Now let's look at 2 practical case studies on Point-to-Point and Publish-Subscribe messaging style.

Point-to-Point Case Study

Domain: Banking
Scenario: Call Center application is requesting customer's savings account details from CASA Core Banking application while serving the on call customer. The diagram below shows end-to-end flow on reference architecture.

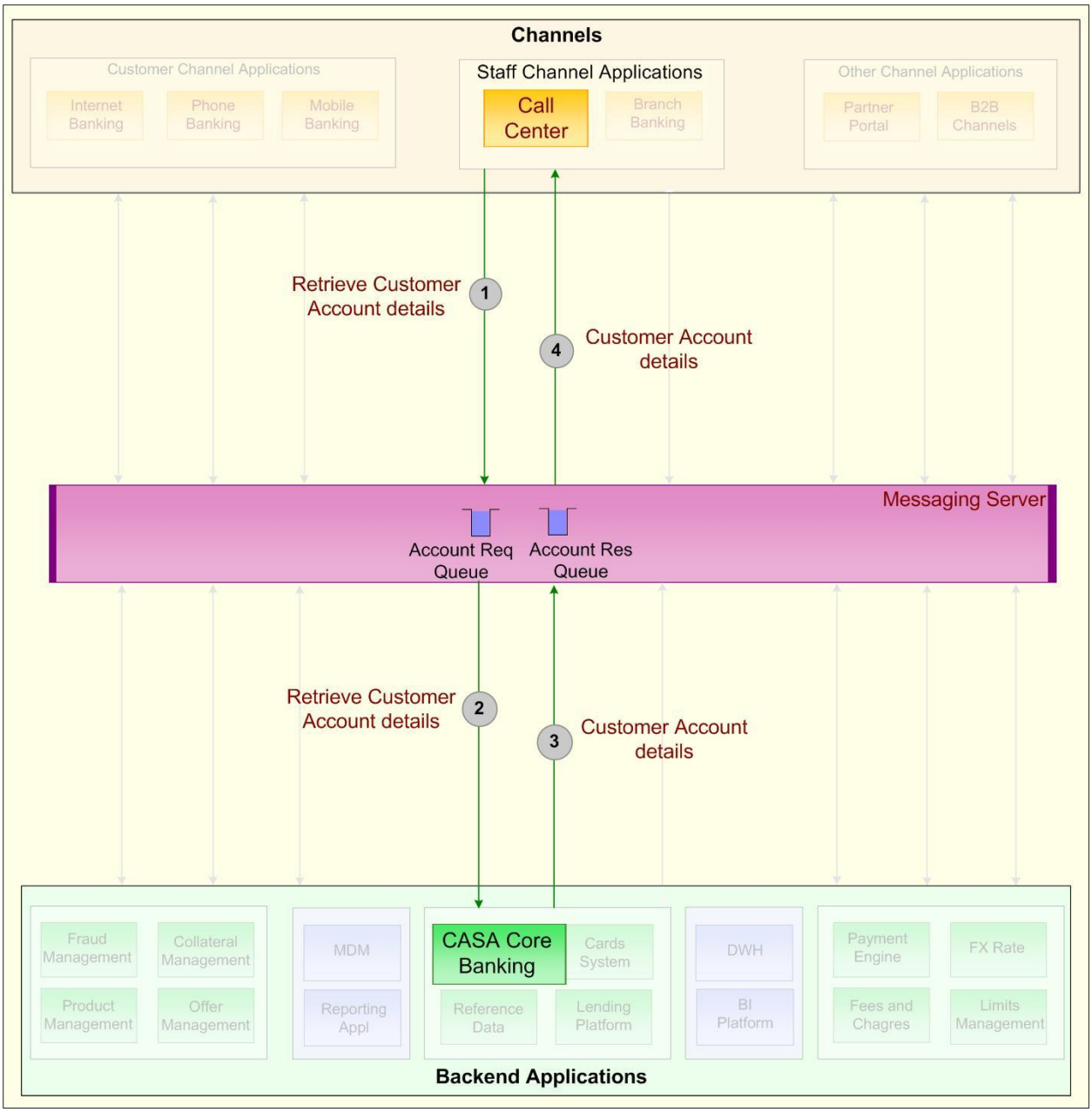

Figure: Enterprise Real-time Integration Reference Architecture: Messaging – Customer account details scenario

Flow -

Step 1: Call Center agent raises the request for Customer Account details through Call Center application while serving the on call customer. Call Center application submits the request for account details retrieval to Account Request queue on messaging server and waits for the response on Account Response queue.

Step 2: CASA Core Banking application consumes the request message and retrieves the Customer Account details from its database.

Step 3: CASA Core Banking application sends back the Customer Account details to Account Response queue on messaging server.

Step 4: Call Center application waiting for reply on Account Response queue consumes the message and displays Customer Account details on the screen/UI.

The same scenarios can be very much mapped to telecom domain where Call Center application can raise request for Bill details to back-end Billing application.

Publish-Subscribe Case Study

Domain: Banking
Scenario: Offer Management application generating and distributing Cashback offer in real-time to various channel applications. This offer then can be communicated to customers through respective channels. For example, offer getting displayed on Internet Banking portal or Call Center agent communicating offer to the on call customer.

The diagram below shows end-to-end flow on reference architecture.

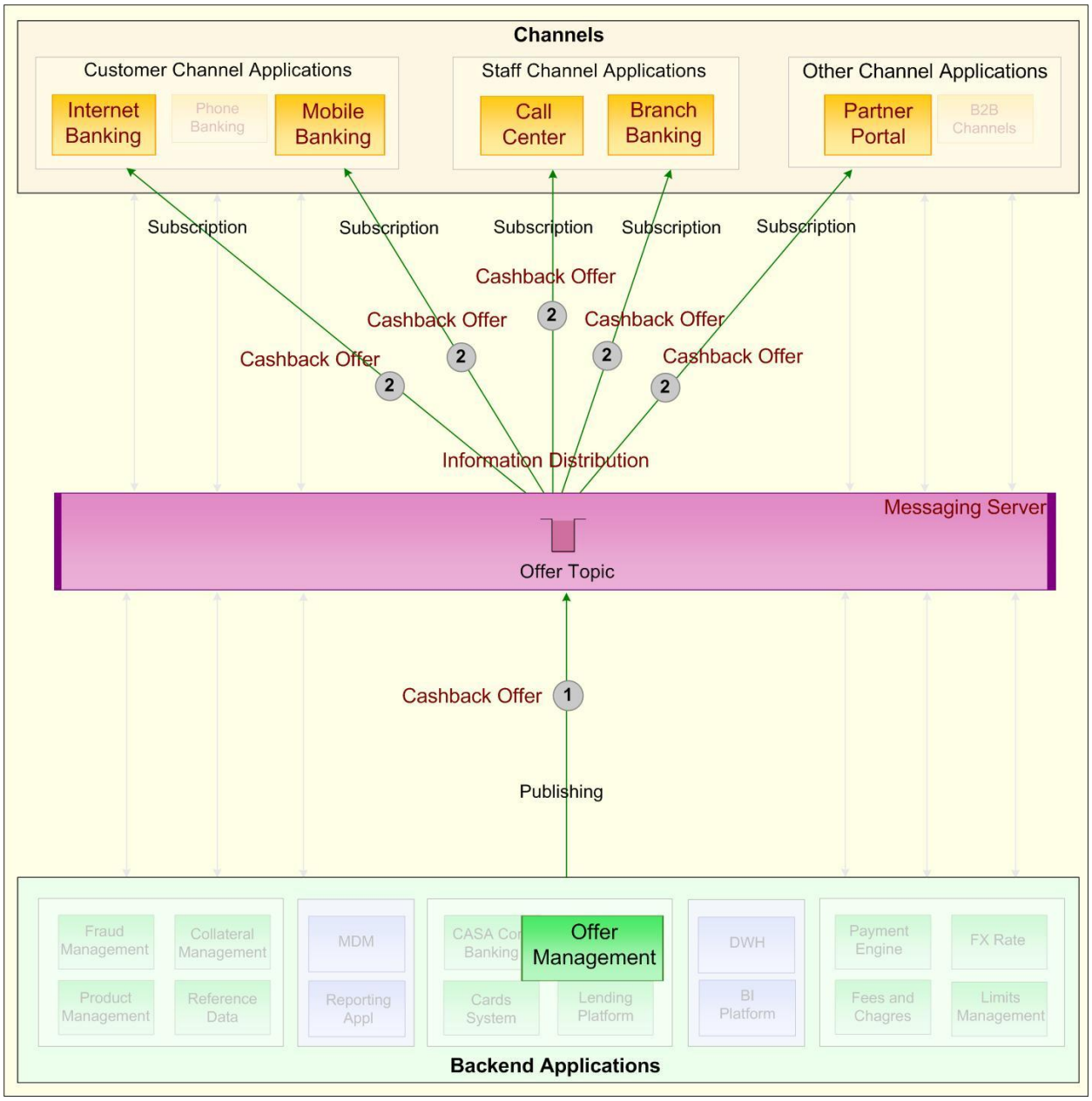

Figure: Enterprise Real-time Integration Reference Architecture: Messaging – Offer Distribution scenario

Flow -

Step 1: Offer Management application generate Cashback offer on cards and publishes the offer information to Offer topic on messaging server.

Step 2: Various channel applications subscribing on Offer topic consumes the Cashback offer message.

The same scenarios can be very much mapped to Telecom domain where offers need to be communicated to customers through various channels (offers on additional talk time, data limit etc.)

In the next chapter let's look at some of the advanced concepts in messaging space.

Chapter 3: Messaging System Advanced Concepts

- Server to Server Bridge
- Federated Architecture
    - Hub and Spoke Topology
    - Mesh Topology
    - Case Study: Manufacturing/Supply Chain Domain
    - Case Study: BFSI Domain
- Fault Tolerance
- Advisory messages and Monitoring
- Anti-Patterns and Performance Consideration

Server to Server Bridge

So far we have discussed the scenarios where multiple clients – producers and consumers are connected to a single central messaging server. However, it is not practical for one messaging server to handle the integration needs of all the Enterprise applications (there could be hundreds of IT applications), due to volume and other considerations. Enterprise applications need to be distributed across multiple messaging servers for better handling of volume and to avoid single point of failure.

While a particular application exchanges data with other applications connected to the same messaging server, it may also need to exchange data with applications connected to another messaging server. And this creates the need to build a bridge/connection between two messaging servers for exchange data with each other. This is a very common requirement in any Organization for building a scalable architecture. Server to Server Bridge creates a connection between 2 messaging servers, the notion of Server to Server Bridges play an important role in creating a network of messaging servers which offers scalability by Design.

This network of messaging servers is must to integrate vast landscape of heterogeneous IT applications which are typically spread across geographical locations and data centers; network of messaging servers enables seamless exchange of data between these applications.

Leading messaging vendors have their own proprietary ways to connect messaging servers and exchange data. It may not be called *Server to Server Bridge,* they may have different terminologies and mechanism for routing/propagating data between messaging severs. But the bottom line is, approach to integrate multiple messaging servers and route/propagate messages across these multiple messaging servers is very much supported by various messaging vendors to build scalable integration solution.

Having said that, for all subsequent discussions, we will stick to the logical notion of Server to Server Bridge and the approach that destinations (queue/topic) with the same name should exist on the connected servers and they should be marked as distributed/routed destinations. This is important to understand the bigger perspective of federated strategy.

Message propagation behavior differs for messages sent on a Queue and messages published on a Topic, let's understand this quickly

Distributed Queues:
For message to get routed between 2 connected messaging servers, queue with the same name need to be defined on both the servers and it should be marked as distributed or routed category.

For example, let's assume there are 2 messaging servers - server A and server B connected via a server to server bridge and distributed queue q1 is configured on both the servers for

exchanging messages; now whenever application sends the message to queue q1 on server A, message will get routed to queue q1 on server B.

Since queue message is destined for only 1 recipient, distributed/routed queue with the same name can exist only in one connected target server. For example, if server A is having distributed/routed queue q1 and server A is connected to server B and server C; then distributed/routed queue q1 can be defined only in one connected target server – server B or server C. Allowing same distributed queue on multiple messaging servers may violate point-to-point messaging model because in a point-to-point model message is destined for a particular Enterprise application which is connected to a dedicated messaging server. (and not destined for multiple Enterprise applications which are spread across different messaging servers). *Having said that, please note that the mechanism and concept may differ from vendor to vendor, but the underlying fact that the message should get forwarded from queue on one server to the corresponding queue on connected server (if configured) remains the same.*

Distributed Topics:

In case of topic there is a difference from distributed queues, distributed topic with the same name can exist on multiple connected target servers. For example, let's assume server A is connected to server B and server C and has distributed/routed topic t1, and there is a need to route the messages published on topic t1 to subscribers connected to server B and server C. In this case distributed topic t1 can be defined on both the servers - server B and server C. This behavior also aligns with the fundamental of Publish-Subscribe model that there could be multiple consumers of the same message (Multiple Enterprise applications spread across different messaging servers may be interested in information being published). *Having said that, please note that the mechanism and concept may differ from vendor to vendor, but the underlying fact that the message should get forwarded from topic on one server to the corresponding topic on connected server (if configured) remains the same.*

Let's have a quick look at server to server bridge case study in brief.

Server to Server Bridge Case Study

Domain: Supply Chain

Scenario: ABC is an electronics product manufacturing company. ABC's Inventory Management application is connected to one of the messaging servers dedicated for internal data exchange. However, all communication to supply chain partners are routed through another messaging server dedicated for B2B communication. Consider the scenario where an Inventory Management application needs to send replenishment request to one of its supplier partner as soon as the inventory level of a particular item goes below threshold limit.

The diagram below depicts this scenario. Inventory management application sends Replenishment request to the *Messaging Server A*. The B2B Gateway application as a

consumer of the message, receives the Replenishment request from *Messaging Server B* to which it is connected. The messages received on queue of *Messaging Server A* are routed to the same queue on *Messaging Sever B,* which is marked as distributed destination.

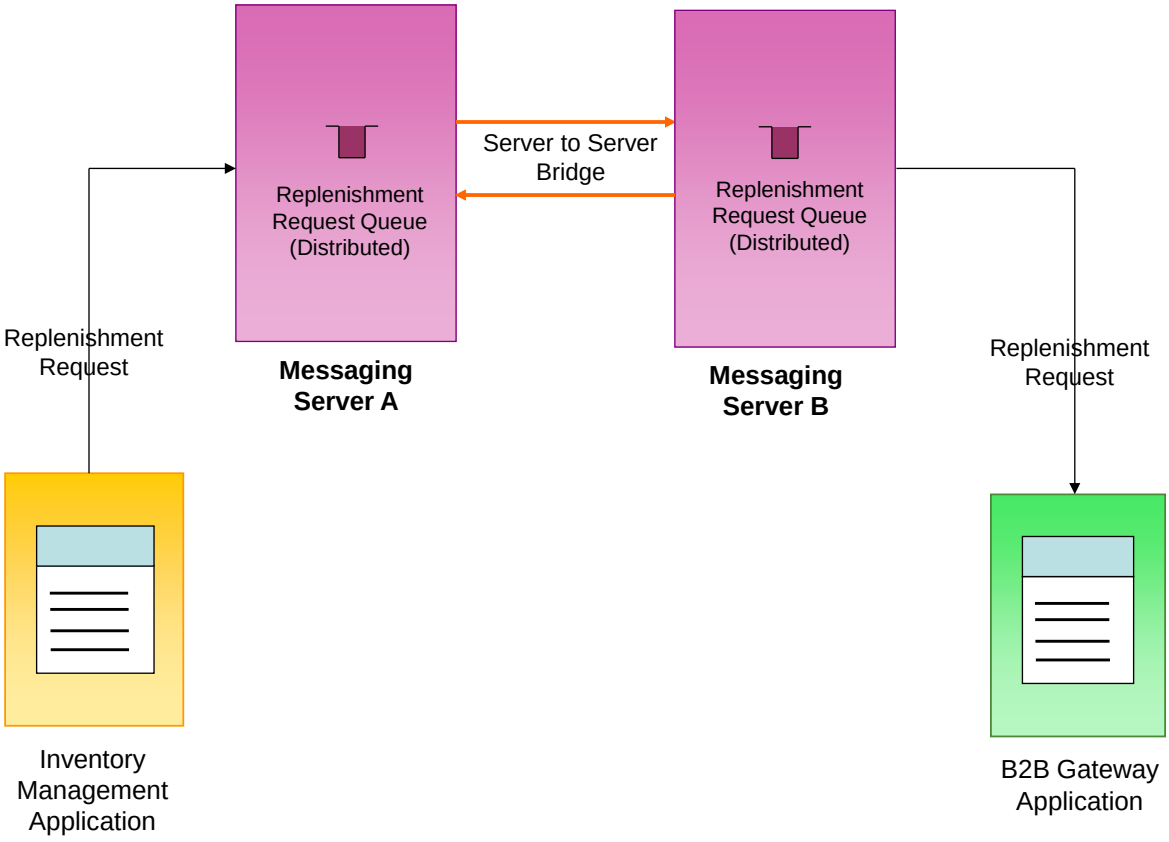

Figure: Server to Server Bridge

The bridge arrangement between 2 messaging servers usually works in reliable mode, what it means is, if bridged server (*Messaging Server B* in this case) is temporarily not available, then *Messaging Server A* retains messages for *Messaging Server B* and messages will be delivered to *Messaging Server B* whenever it is available again, this is to avoid data loss between connected servers. Bridge can be built in both the direction. Messages can flow from *Messaging Server A* to *Messaging Server B* and vice versa.

Now having understood the fundamentals of Server to Server Bridge and Message propagation/routing between servers, let's look into topologies that can to be used to build scalable integration solution.

Federated Architecture

As discussed in the earlier section on Server to Server Bridge, it is not practical for one messaging server to integrate all the applications of the Organization. Each region within any Global Organization operates as a separate individual mini Organization and the region itself contains host of applications which needs integration with each other to exchange data. It may not be possible to scale one messaging server vertically to handle the volume coming from all Enterprise applications. So the first thing is to design overall Enterprise integration strategy in such a way that it is flexible and scalable by design. Messaging server needs to be scaled horizontally and appropriate federation strategy should be in place to build Enterprise level integration solution.

There are 2 common topologies which can be followed in defining federated messaging architecture

- Hub and Spoke
- Mesh

Above are the topologies which defines how messaging server arrangement looks like from scalability perspective. Irrespective of the topology, messaging servers will be connected to each other using Server to Server Bridges.

Let's take a look at these topologies one by one

Hub and Spoke Topology

In this style of federated strategy, messaging servers are not connected to each other directly; however, they are connected to each other via a central messaging server. This style of arrangement is called Hub and Spoke topology. Please note that in a JMS based messaging architecture, each messaging server itself fundamentally operates in Hub and Spoke model where various clients connect to messaging server for the purpose of sending or consuming messages. Messaging server acts as a hub and all the producers and consumers of message act as spokes. Here we are talking about the pattern/arrangement in which the messaging servers themselves are connected to each other in hub and spoke style for exchanging messages between them. The diagram below depicts Hub and Spoke topology; in this case, the Central Routing Hub/Gateway acts just as a pass-through with no producers and consumers directly connected to it.

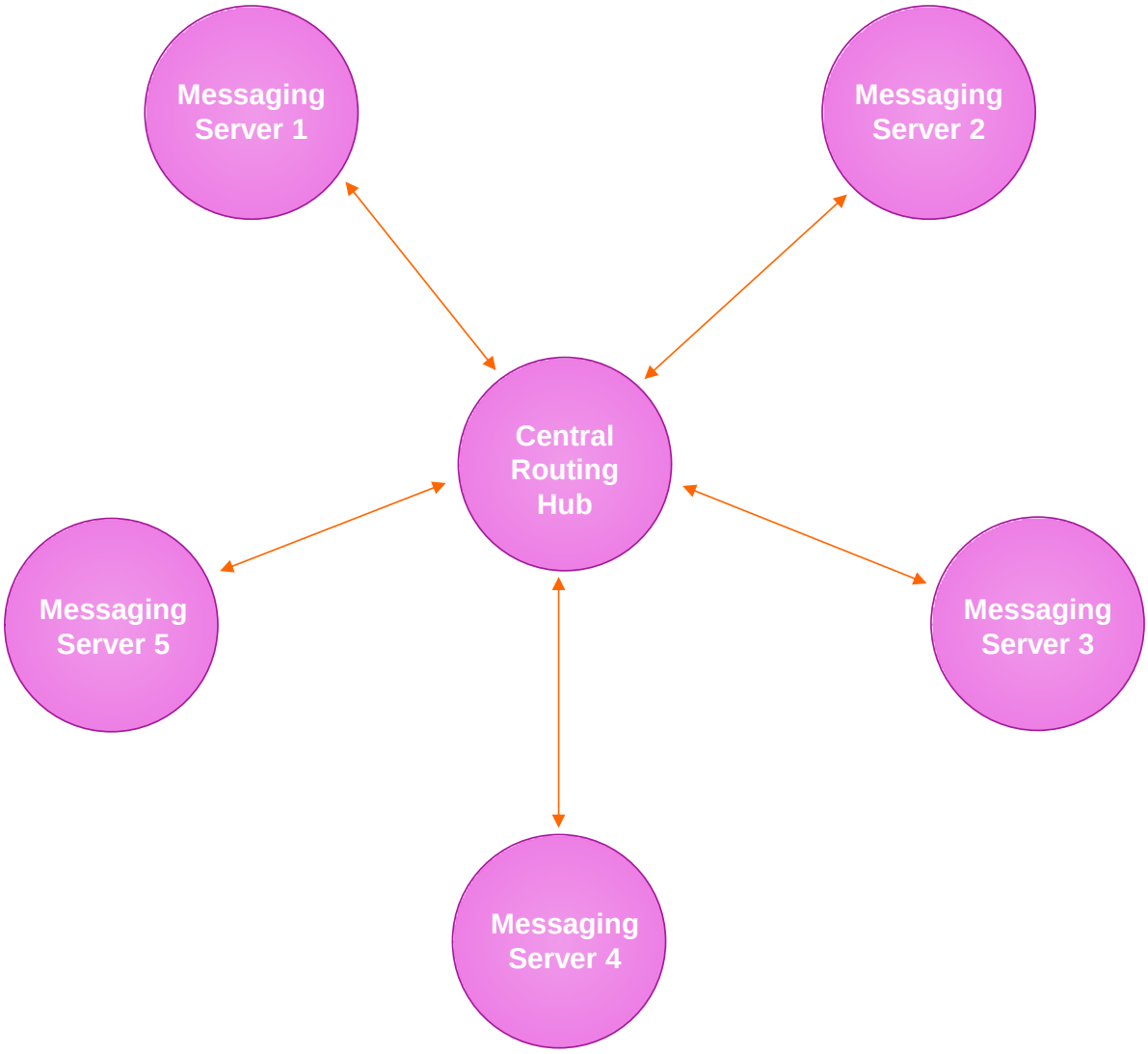

Figure: Hub and Spoke Topology of Messaging Servers

Advantages:

- All messages are channeled through central routing hub, any change to other server will not impact the routing/bridge configuration of rest of the servers as routing/bridge is configured to central routing hub.
- Low maintenance.

Disadvantages:

- Single point of failure for server to server communication, if central routing hub goes down due to some mis-configuration or hardware issue, communication between all the servers will be impacted.
- Latency for message travel from producer to consumer due to additional hop.

If this topology of federation is to be followed, then it should not result in a situation where heavy message traffic flows via central routing hub. For example, if application1, application2, application3 and application4 involve significant amount of message exchange between them, then they should not be distributed across different messaging servers connected via central routing hub. This topology is suitable to connect applications which involve limited traffic flow through central routing hub; thus reducing business impact in case of central routing hub failure. If there is a heavy traffic flowing through central routing hub, then any issue with central routing hub may result in larger Business impact.

Mesh Topology

In this style of federated strategy, messaging servers are connected to each other directly; this style of arrangement is called Mesh topology. Mesh topology is more suitable for applications which involves significant traffic flow between them. In this topology there is no central messaging server mediating messages between other messaging servers. The diagram below depicts Mesh topology.

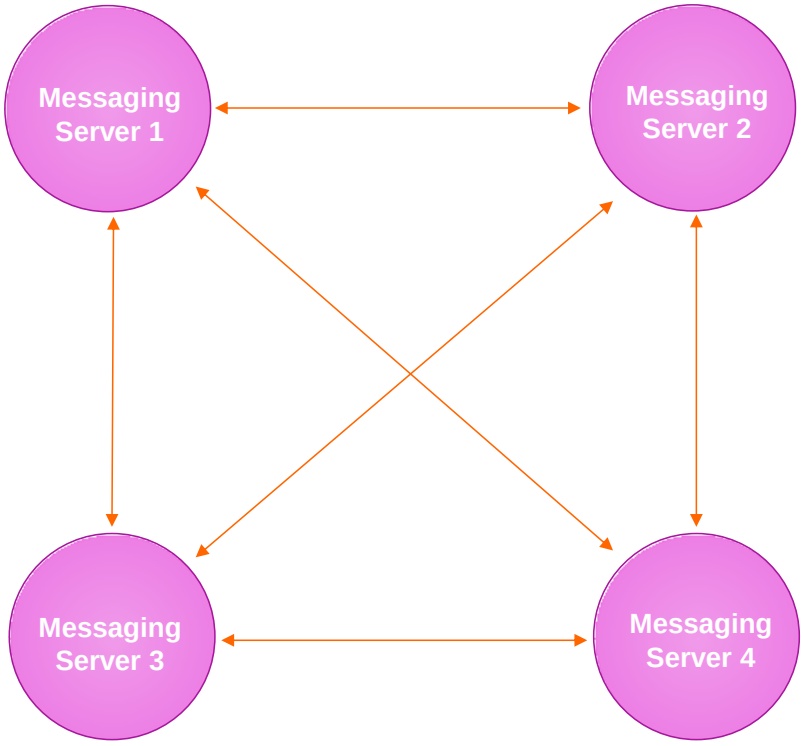

Figure: Mesh Topology of Messaging Servers

Advantages:

- The advantage of choosing this topology is, if one of the servers is impacted or goes down, communication between rest of the messaging servers are not impacted; there is no single point of failure or bottleneck.
- Offer better SLA and performance as there is no additional hop, messaging servers are connected to each other directly for exchanging messages.

Disadvantages:

- The drawback of choosing this topology is that the maintenance cost can go higher as number of connection grows. If messaging infrastructure needs further scalability, this will be overhead.

This is a good topology to implement if messaging servers to be connected are less in numbers.

Now let's look at few case studies from different business domains to understand how Enterprise level federated messaging architecture can be defined using above topologies.

Case Study: Manufacturing/Supply Chain Domain

Requirement/Challenge:
Electronics product manufacturing Company ABC need to integrate various ERP and Channel applications within business units as well as applications spread across multiple business units. The most fundamental requirement is that, all these applications should be loosely coupled to support Company's ongoing journey of legacy modernization so that impact to interfacing application is minimal.

Additionally, company has following requirement on integration from scalability, performance and Business perspective

1. With growing dealer/retailer/supplier base, integration solution should support Year-On-Year volume growth of 15%.
2. Quicker exchange of data between cohesive applications giving faster SLAs for end-to-end execution of Business Processes.
3. Unforeseen temporary failure of integration component should not lead to major Business impact.
4. Better service response for channel applications involving customer engagement.

Solution:

The most basic requirement of loosely coupled integration environment supporting legacy modernization with minimal impact to interfacing application makes the case for messaging solution without any doubt. And now let's look at solution for rest of the critical requirements on scalability, business impact, faster SLA etc.

1. With growing dealer/retailer/supplier base, integration solution should support Year-On-Year volume growth of 15%.
   Answer: Use of multiple messaging servers (as against single messaging server) to provide better scalability (ability to handle growing volume) by design.

2. Quicker exchange of data between cohesive applications giving faster SLAs for end-to-end execution of Business Processes.
   Answer: Connecting cohesive-interrelated applications which involves good amount of data exchange using same messaging server.

3. Unforeseen temporary failure of integration component should not lead to major Business impact.
   Answer: Distributing Business Critical applications across multiple messaging servers and not connecting all critical applications to same messaging servers.

4. Better service response for channel applications involving customer engagement.
   Answer: Reduce hops between Channel applications and back-end provider applications.

With above considerations in mind, let's define solution for integrating key ERP and back-end applications with Channel applications.

Approach:

- Use multiple messaging servers for better scalability – 2 channel messaging servers for Channel applications and 2 provider messaging servers for back-end ERP applications.
- Distribute 2 key high traffic channels (call center application and self-service portal) across 2 channel messaging servers to reduce the business impact in case of unforeseen failure of a particular channel messaging server (due to issues related to mis-configuration /deployment /client connection loop, which may also impact highly available channel server).
- Connect related Order-Billing applications to one provider messaging server and supply chain management (SCM) applications to another provider messaging server.
- Connect channel messaging servers with provider messaging servers in MESH topology (direct connectivity) for better response SLA and performance.

The diagram below depicts the overall integration solution based on MESH topology.

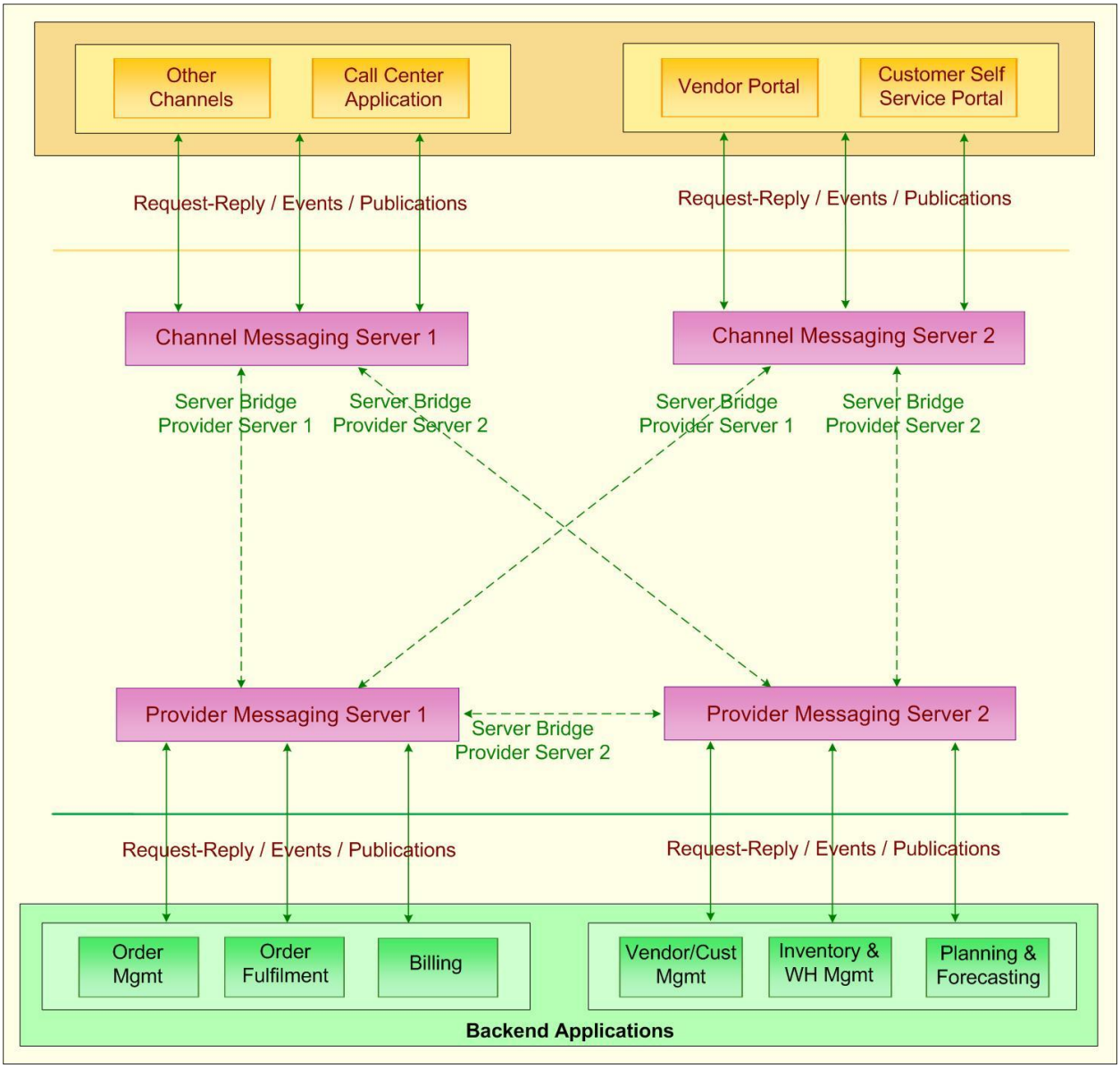

Figure: Mesh Topology: Supply Chain Case Study

So as depicted in the diagram, channel applications are spread across different channel messaging servers and back-end provider applications are connected to different provider messaging servers. Channel messaging servers are connected to provider messaging servers using Server Bridges. Please note that the connection between *Channel Messaging Server 1* and *Channel Messaging Server 2* is not shown in the diagram above, but if there is a use case to exchange data between channel applications, channel messaging servers can always be connected.

Channel Messaging Server 1 - Call Center and other channel applications
Channel Messaging Server 2 - Vendor Portal, Customer Self Service Portal
Provider Messaging Server 1 – Order Management, Order Fulfillment, Billing (cohesive applications)
Provide Messaging Server 2 – Inventory and Warehouse Management, Planning and Forecasting, Vendor/Customer Management (cohesive applications)

Though the messaging servers are named as Channel and Provider Messaging Servers to align them logically with channels and back-end applications, they are just messaging servers dedicated to set of applications.

Now let's look at couple of scenarios to understand end-to-end flow on federated architecture.

Scenario 1:
Submitting Purchase Order (PO) details from Call Center or Customer Self Service portal application to Order Management application for processing.

The diagram below shows the routing of PO event from *Channel Messaging Server 1* to *Provider Messaging Server 1* (target server) and *Channel Messaging Server 2* to *Provider Messaging Server 1* (target server). And as discussed in 'Distributed Queues' section, distributed/routed queue (in this case PO event queue) should be available on both the connected servers for message to be routed successfully.

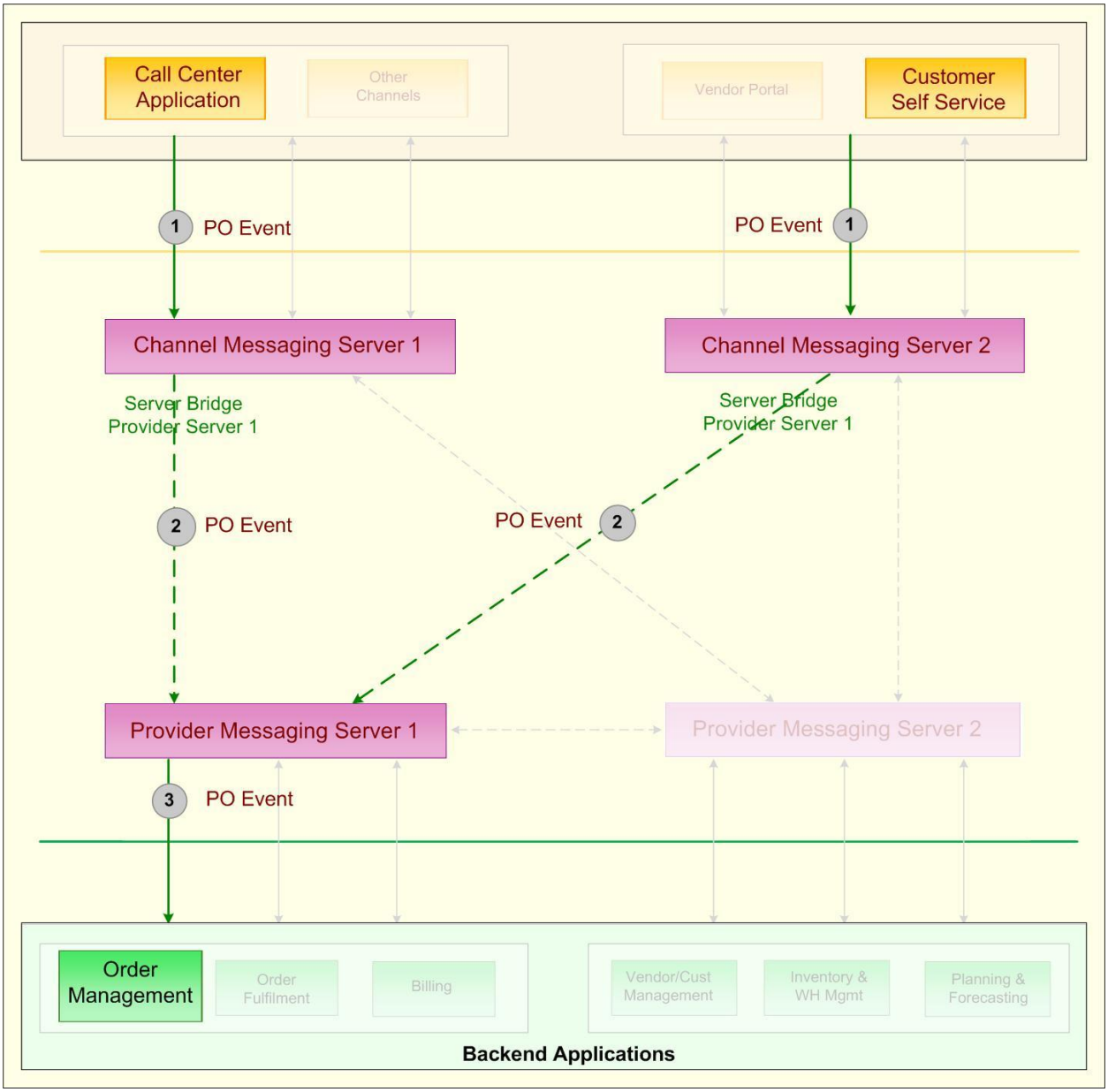

Figure: Mesh Topology: PO event

Scenario 2:
Call Center application performing ATP check (Available-To-Promise check) against Inventory Management system while booking the order for on call customer.

The diagram below shows routing of ATP Check request from *Channel Messaging Server 1* to *Provider Messaging Server 2* and routing of ATP Check response from *Provider Messaging Server 2* to *Channel Messaging Server 1*. Distributed/routed request and response queues (in this case ATP Check request queue and ATP Check response queues) should be available on both the connected servers for message to be routed successfully.

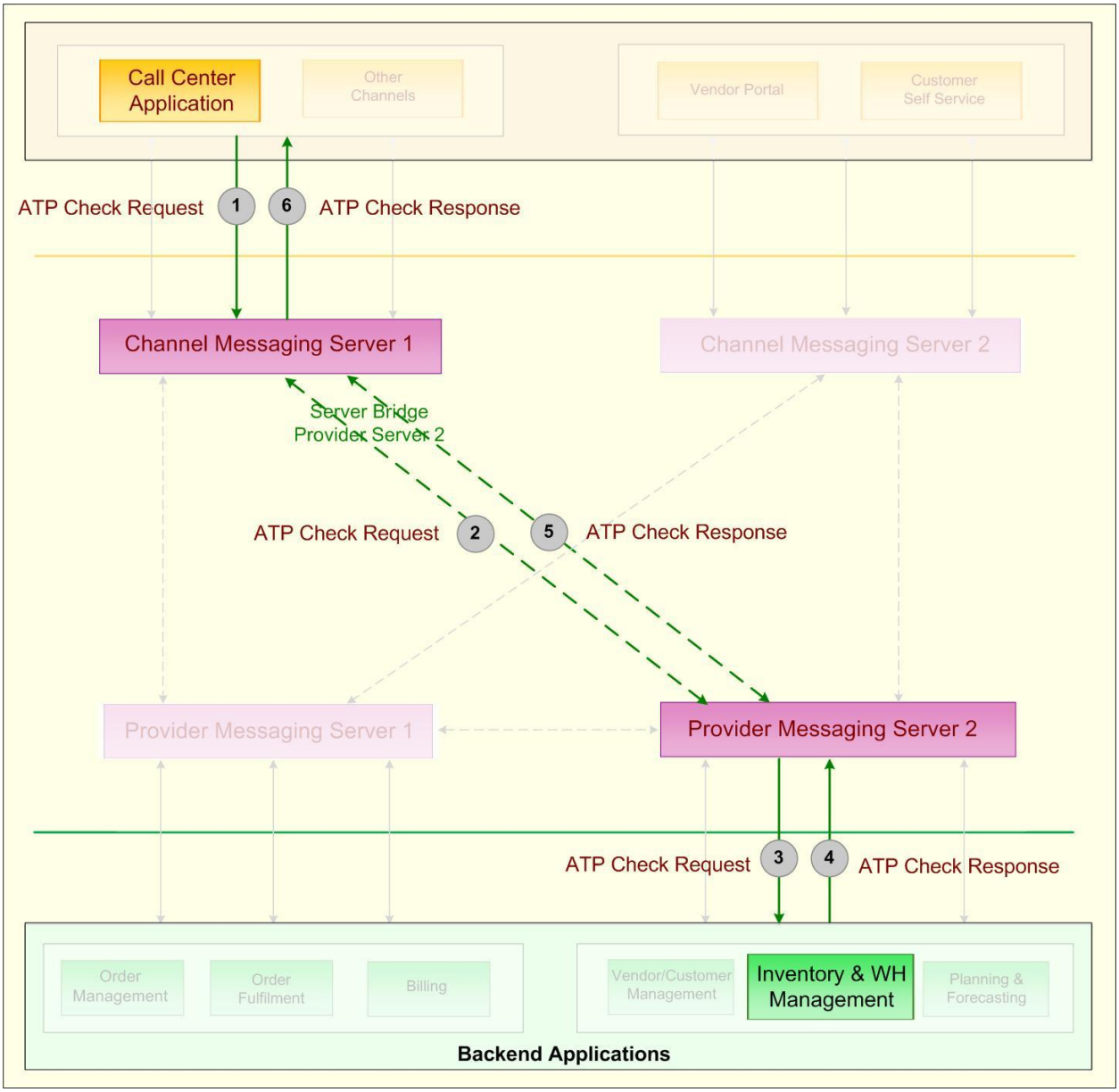

Figure: Mesh Topology: ATP Check request

There is an important aspect to notice here. Source application need not write message routing logic explicitly in the code, it simply sends the message to its local dedicated messaging server which internally takes care of forwarding it to appropriate target messaging server based on distributed/routed queue configuration; where the message gets routed is completely transparent to the source application. This kind of message routing between messaging servers is very common approach for 2 simple reasons

- Routing all traffic coming from application (and related applications) through a dedicated local messaging server reduces the number of connections across the network.
- Dedicated messaging server can be configured for intelligent routing of messages to appropriate target messaging server in the network where the Service Provider is connected, this is completely transparent to the source application.

In addition to above scenarios related PO Event and ATP check, below are few more examples of data exchange between various applications

- Order Status events from Order Fulfilment (Order Shipped) application to Call Center application (routing from *Provider Messaging Server 2* to *Channel Messaging Server 1*).
- Order Confirmation event from Order Management application to Order Fulfilment application (via same *Provider Messaging Server 1*) for shipping the order (no server to server bridge involved for these cohesive applications).
- Order Billing event from Order Fulfilment application to Billing application (via same *Provider Messaging Server 1*) for generating the invoice.

After understanding the use of MESH topology, now let's extend this case study to see use of Hub and Spoke topology and its role in connecting applications which involves less traffic flow between them.

As the number of applications and respective messaging server grows, it is really difficult to continue using Mesh topology and build direct connectivity between these messaging servers. It leads to more complex integration landscape which will be difficult to maintain. The approach here should be to use mix topology to build scalable architecture. The simple thumb rule is, integrate applications involving heavy data exchange using Mesh topology and connect applications involving limited data exchange using Hub and Spoke topology.

The diagram below shows the use of Mix topology (Mesh and Hub-Spoke) from a bigger perspective.

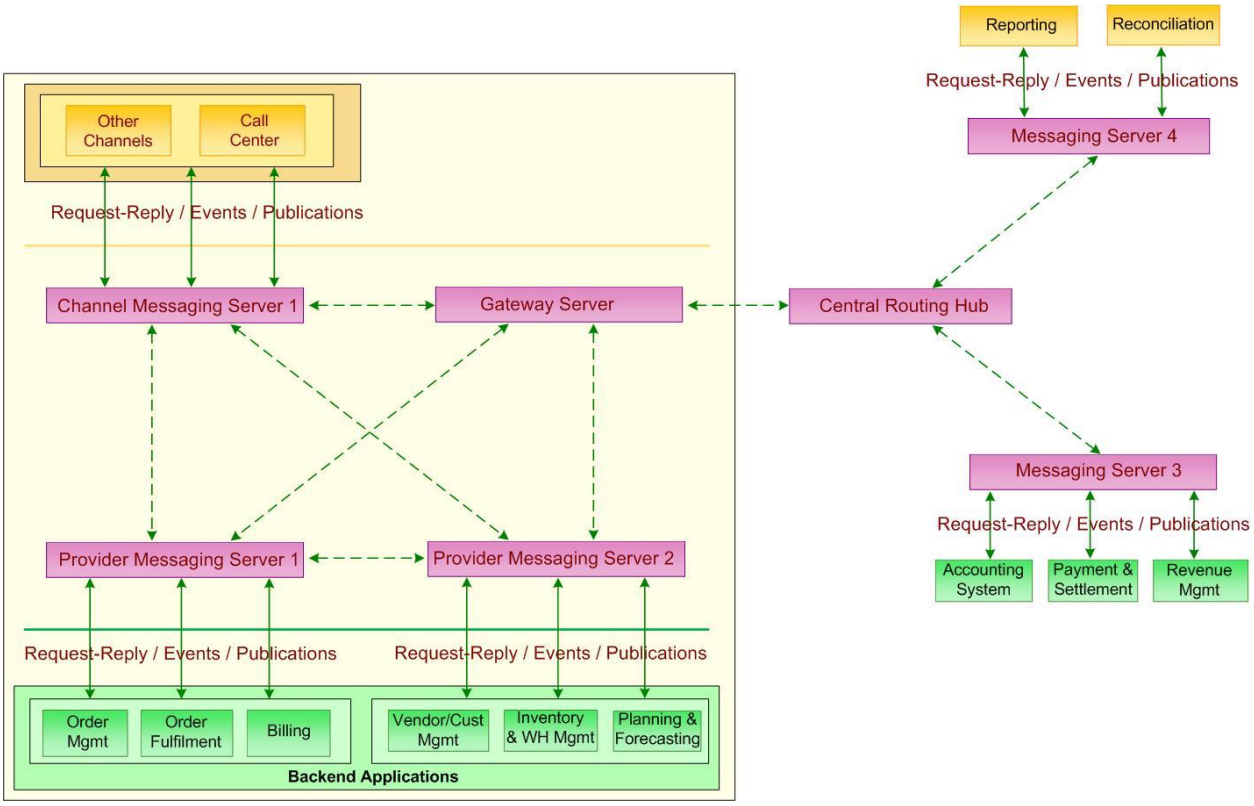

Figure: Mix Topology

As depicted in the diagram, Financial management applications (Accounting, Payment and Settlement, Revenue Management etc.) are integrated with Order Management-Billing via central routing messaging server using Hub and Spoke topology.

Scenario 3:
Billing application posting Accounting Entry to Accounting system (recording receivables) once the invoice is generated for dealer/retailer.

The diagram below depicts routing of Accounting Entry event
Provider Messaging Server 1 → *Gateway Server* → *Central Routing Hub* → *Messaging Server 3*. Usually, propagation of event across multiple messaging servers is supported through *Topic* rather than *Queue*. As discussed in 'Distributed Topics' section, distributed/routed topic (in this case Account Posting topic) should be available on all the servers for message to be routed successfully.

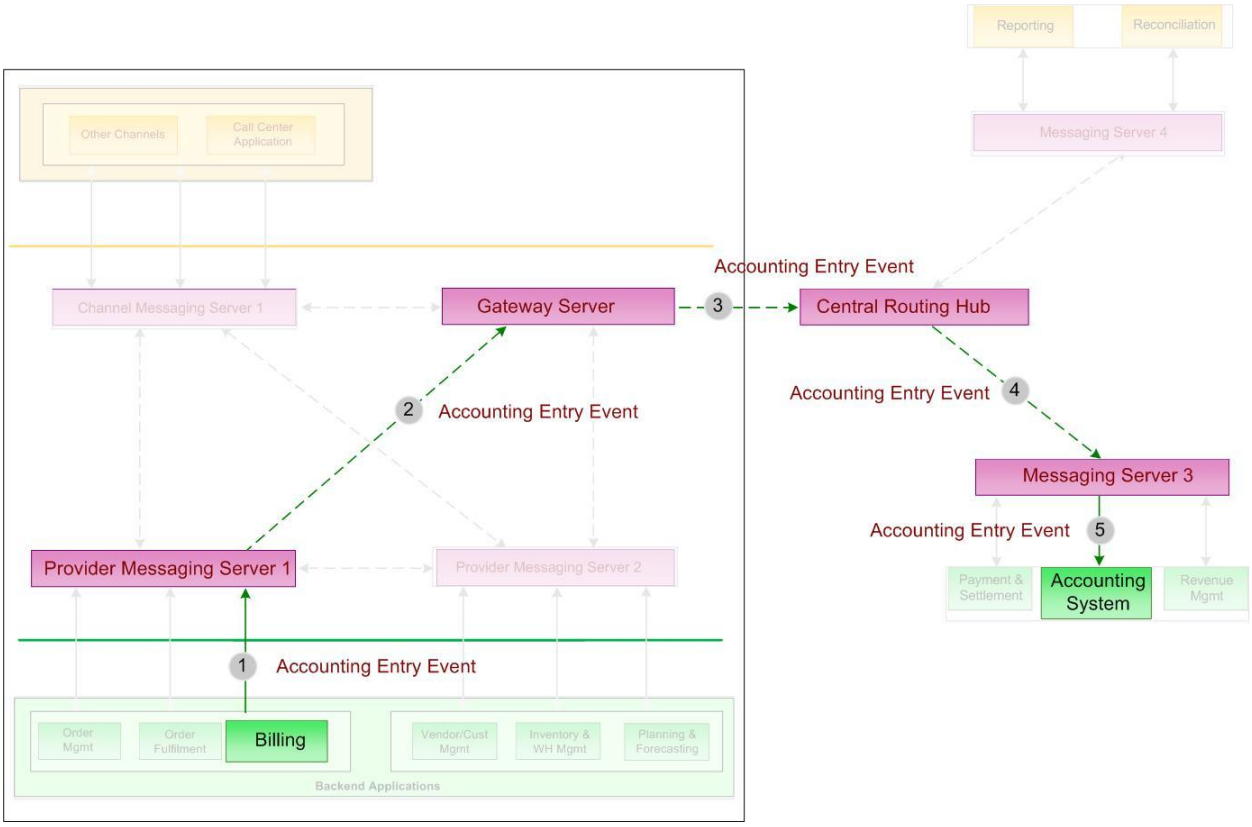

Figure: Mix Topology: Accounting Entry Event

Case Study: Banking Domain

Now let's look at the case study from Banking domain. Let's assume the requirements of Year-On-Year volume growth, less impact to Business, faster SLAs, scalability are very much similar to Supply Chain domain case study.

With that let's define solution for integrating various Channel applications of Bank with back-end core systems.

Approach:

- Use multiple messaging servers for better scalability – 3 channel messaging servers for Channel applications and 3 provider messaging servers for back-end core systems and supporting systems.
- Distribute 3 key Channels (staff channel applications - *call center, branch*; customer self-service channel application – *internet, mobile banking and Partner channels*) across 3 channel messaging servers so as to reduce the business impact due to failure of a particular channel messaging server.
- Distribute back-end applications across 3 provider messaging servers.

- Connect channel messaging servers with provider messaging servers in MESH topology (direct connectivity) for better response SLA and performance.

The diagram below depicts the overall integration solution based on MESH topology.

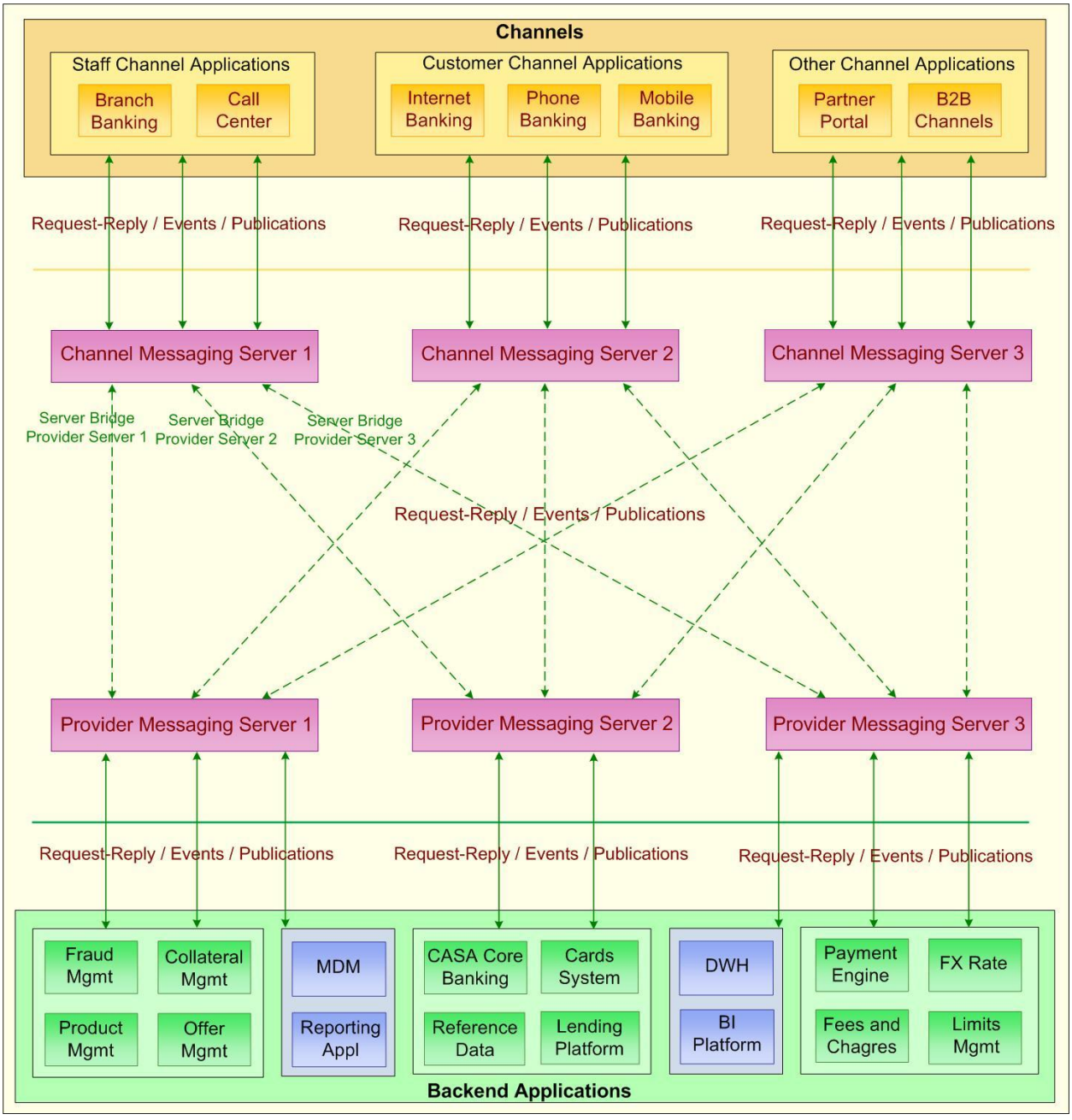

Figure: Mesh Topology: Banking Case Study

As depicted in the diagram above, key customer interaction channels are well distributed across different messaging severs, thus reducing impact to customers in case of unforeseen failure of one of the channel messaging servers.

Now let's look at couple of scenarios we have already seen in reference architecture section. Those scenarios were discussed considering a single messaging server just to provide basic understanding of flow, but in reality, with federated architecture in place, the flow will be different with messages getting routed through multiple messaging servers.

Scenario 1 (Point-To-Point Scenario):
Call Center application raising request for customer savings account details from CASA Core Banking application while serving the on call customer.

The diagram below depicts the routing of Customer Account details request from *Channel Messaging Server 1* to *Provider Messaging Server 2* and routing of Customer Account details response from *Provider Messaging Server 2* to *Channel Messaging Server 1*. And as discussed in 'Distributed Queues' section, distributed/routed request and response queues (in this case Account Request and Account Response queues) should be available on both the connected servers for the message to be routed successfully.

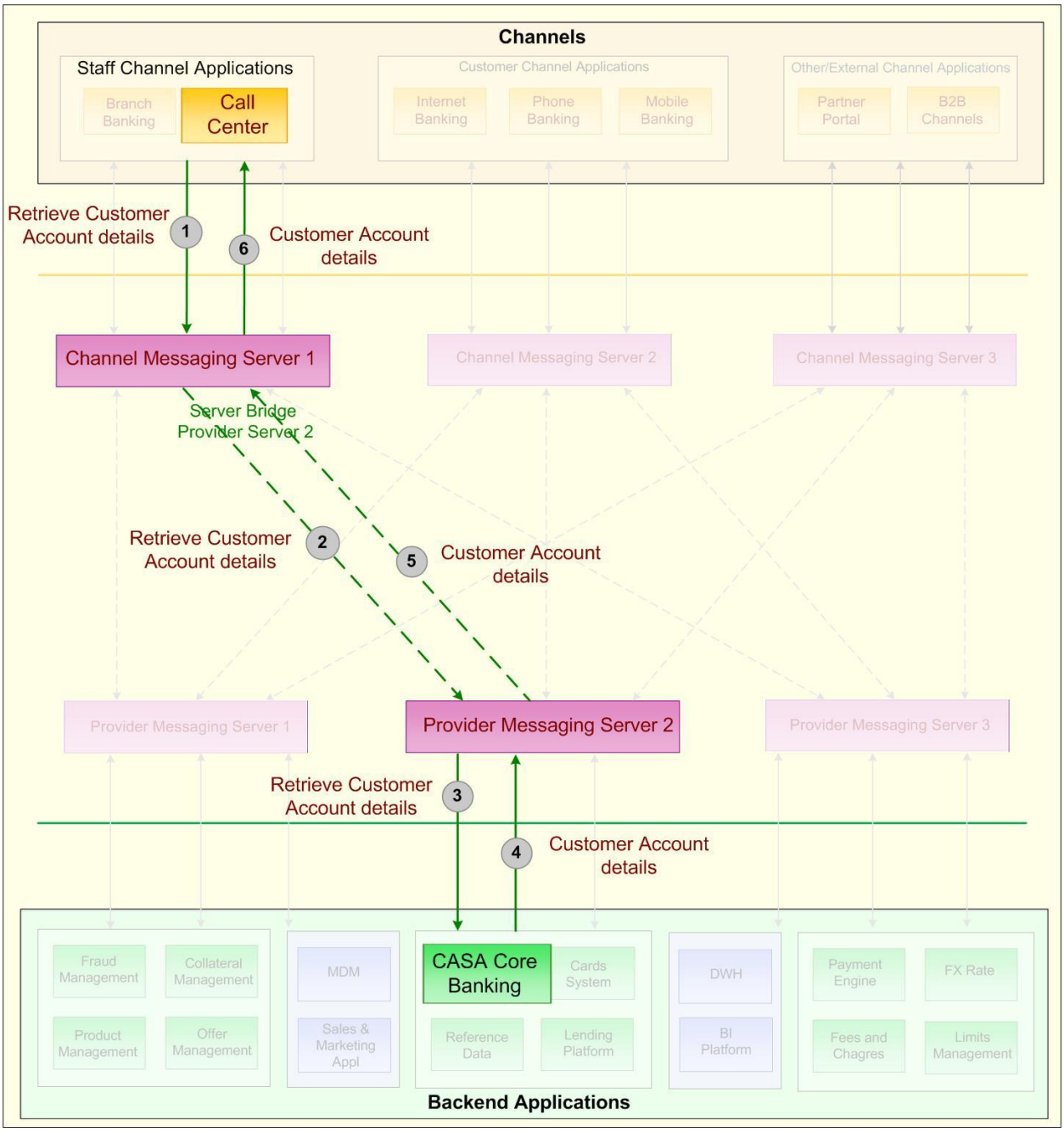

Figure: Mesh Topology – Customer Account details request

Scenario 2 (Publish-Subscribe Scenario):
Offer Management application generating and distributing Cashback offers on real-time basis to various subscriber applications. These offers then can be communicated to customers through respective channels. For example, Cashback offer getting displayed on Internet Banking portal or Call Center agent communicating the same to the on call customer.

The diagram below depicts distribution of Cashback offer information from *Provider Messaging Server 2* to *Channel Messaging Server 1*, *Channel Messaging Server 2* and *Channel Messaging Server 3*. It is important to note that message is published on one server and respective subscribers are consuming message from the server they are connected to. Distributed/routed topic (in this case Offer topic) should be available on all the servers for the message to be routed successfully.

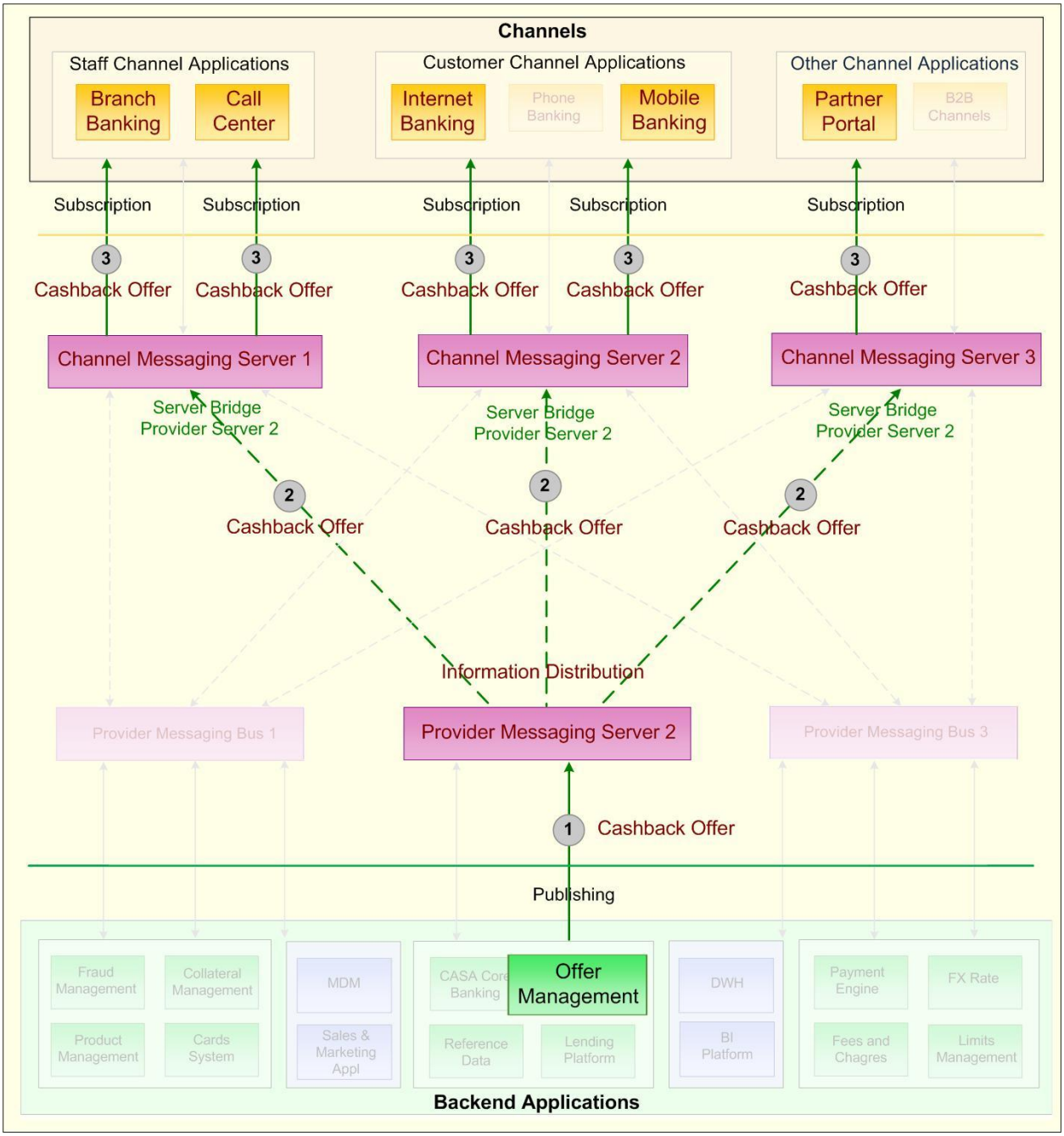

Figure: Mesh Topology – Cashback Offer Distribution

Conclusion:
Deciding on the topology is not straight forward; assessment plays a key role in choosing the right topology.

So as a general guideline, following aspects needs to be considered while defining federated strategy

- Consideration for scalability

  Multiple Messaging Server – One messaging server is not sufficient to integrate hundreds of Enterprise applications. Multiple messaging servers are required to integrate these applications; the overall message traffic needs to be well distributed across multiple servers providing better scalability by design. Adopt an approach of allocating a messaging server to set of related/cohesive applications.

  Volume and Future Growth: While allocating a single messaging server to multiple cohesive applications might be the general approach, message volume and future growth plays vital role on deciding the actual number of messaging servers required for cohesive applications. For example, a high volume CRM application or Payment application may need a dedicated messaging server. The bottom line is, not to connect all cohesive high volume applications to a single messaging server. Up-front scalability consideration is very important while defining sustainable federated architecture strategy.

- Consideration for performance

  It might be a good idea to connect applications involving high data exchange using Mesh topology (direct connectivity of messaging server); this will also ensure improved overall SLAs by eliminating additional hop. Use Hub and Spoke topology to connect applications which involves less data exchange between them.

- Consideration for Business impact

  Distributing business critical applications across multiple messaging servers is key strategy to reduce risk to overall Business. As discussed in the case studies, key channels should be distributed across different messaging servers. If due to some unforeseen failure, messaging server supporting one of the key channels goes down, this impacts only that particular channel; rest of the channel's communication with back-end applications is not impacted as they are connected using different messaging servers. This ensures that Organizations Business is still up and running through other non-impacted channels and there is no single point of failure.

Case studies discussed above provides an indicative guidelines. What topology suits Business Domain (Hub and Spoke or Mesh or Mix) depends on the requirements; proper assessment is required to arrive at the Enterprise level federated architecture topology.

Fault Tolerance

Overview

In general terms, fault tolerance provides an environment by which there is no interruption to the services provided in case of failure of the system. In this section, we will discuss fault tolerance from the perspective of messaging server. So essentially, if one messaging server goes down, other messaging server assures continuation of the services to all the Message Producer and Message Consumer applications. This can also be seen as Highly Available environment.

Typically in messaging, high availability behavior is achieved by adding two or more messaging servers in a cluster with Active-Passive arrangement (also referred as Primary-Backup or Master-Slave). In Active-Passive configuration, active server is the primary server to whom all the clients are connected, passive server is the standby server which is kept ready to provide services in case active server goes down due to hardware failure or some other reason.

Shared State

For the efficient failover, the active server and the passive server must share the same state.

State information typically includes

- PERSISTENT messages
- Information on Client connections to the active server

What it means is, PERSISTENT messages are available across all the servers in a cluster (active and passive servers); so that when the active server goes down and passive (standby) server takes the charge, it can deliver undelivered PERSISTENT messages to consumers.

So essentially, state information must be stored in shared store which could be:

- Shared File System OR
- Shared Database

In case of Shared File System, all the messaging servers in a cluster are configured to point to the same directory, for example, *//SharedFileDir/SharedServerData*

Irrespective of whether it is file storage or database storage, most of the shared storage solution supports the concept of exclusive lock and allows only one server to be in possession of the lock.

Normally, active server keeps sending heartbeat signal to passive servers indicating its active status. When passive servers do not receive heartbeat signal from active server, they attempt to take lock of shared state to assume the role of active server.

Active-Passive Scenario

To achieve fault tolerance, two or more servers are added in a cluster; whichever server starts first and able to grab the exclusive lock on shared state becomes the active server, rest of the servers assumes the role of passive-standby servers. The diagram below shows the scenario where *Messaging Server S1* is active server and *Messaging Server S2* and *Messaging Server S3* are passive servers.

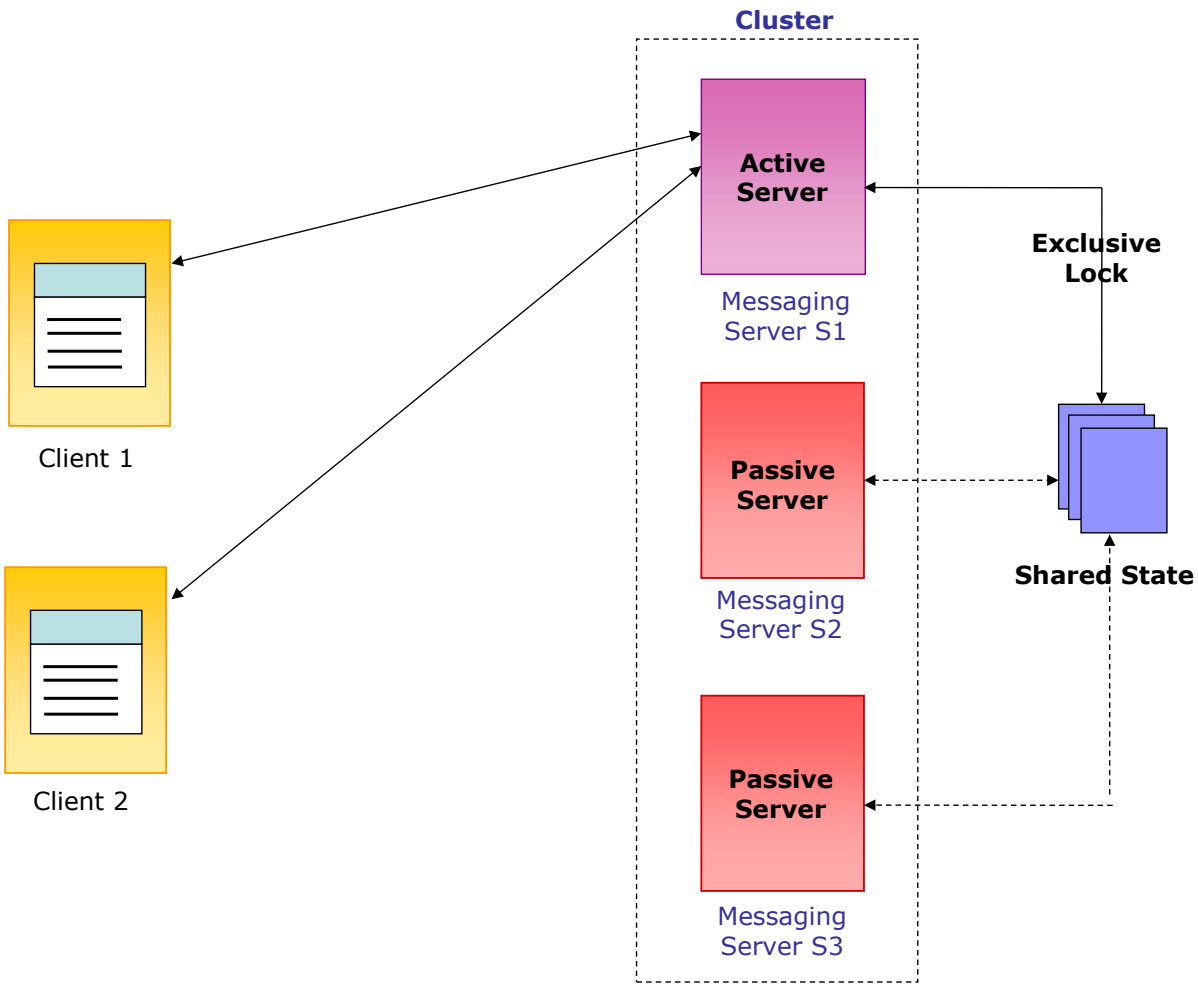

Figure: Active-Passive Arrangement

Active Server Failure and Role Swap

If active server fails, the lock on shared state is released automatically. Passive servers race to grab the lock and whichever server gets the exclusive lock on the shared state first becomes the active server, other servers continue to act as passive servers. The diagram

below shows the scenario where *Messaging Server S2* becomes the active server post failure of *Messaging Server S1* and *Messaging Server S3* continue to act as a passive server.

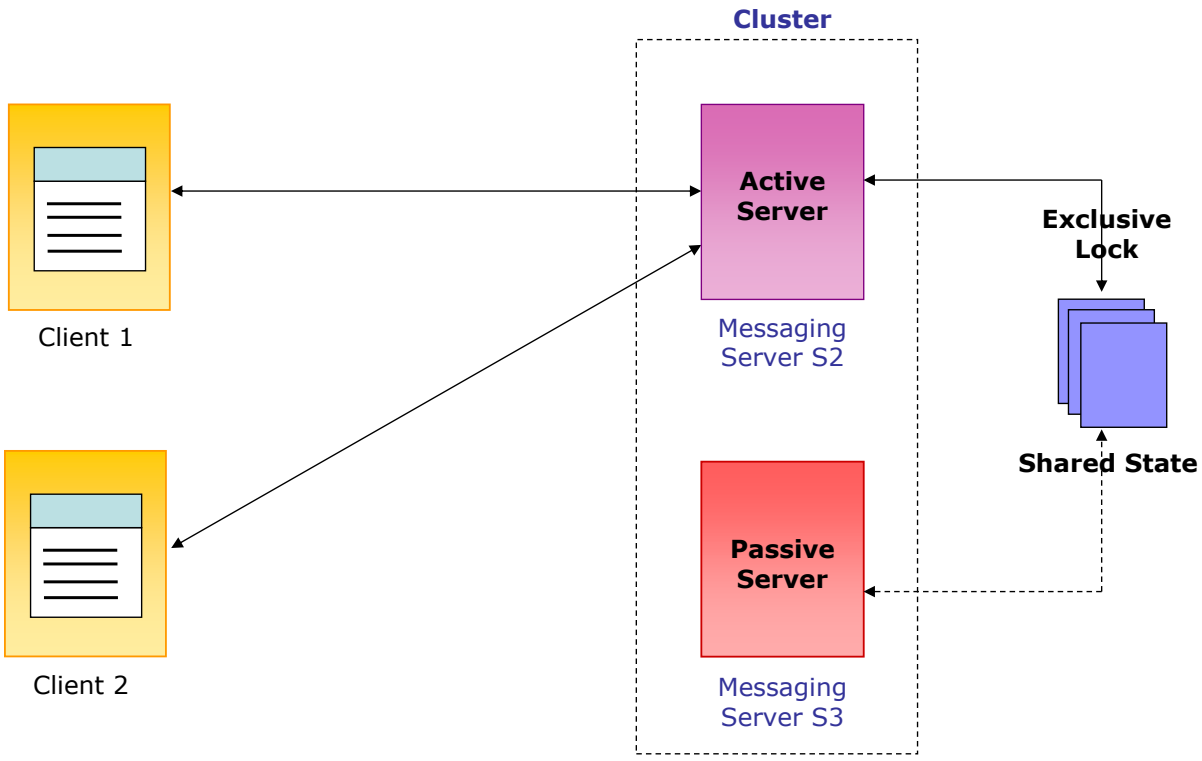

Figure: Failover Scenario

The clients of *Messaging Server S1* are automatically transferred to *Messaging Server S2. Messaging Server S2* will deliver the PERSISTENT messages, if any, to the re-connected consumer clients. Consumer clients who were supposed to receive PERSISTENT messages from *Messaging Server S1*, but could not get it after *Messaging Server S1* failure, will now receive the messages from *Messaging Server S2*. Clients can seamlessly failover from one messaging server to another server and keeps receiving or sending messages without interruption.

Now if the old active server *Messaging Server S1* restarts after failure, it acts as a passive server. The diagram below depicts this scenario.

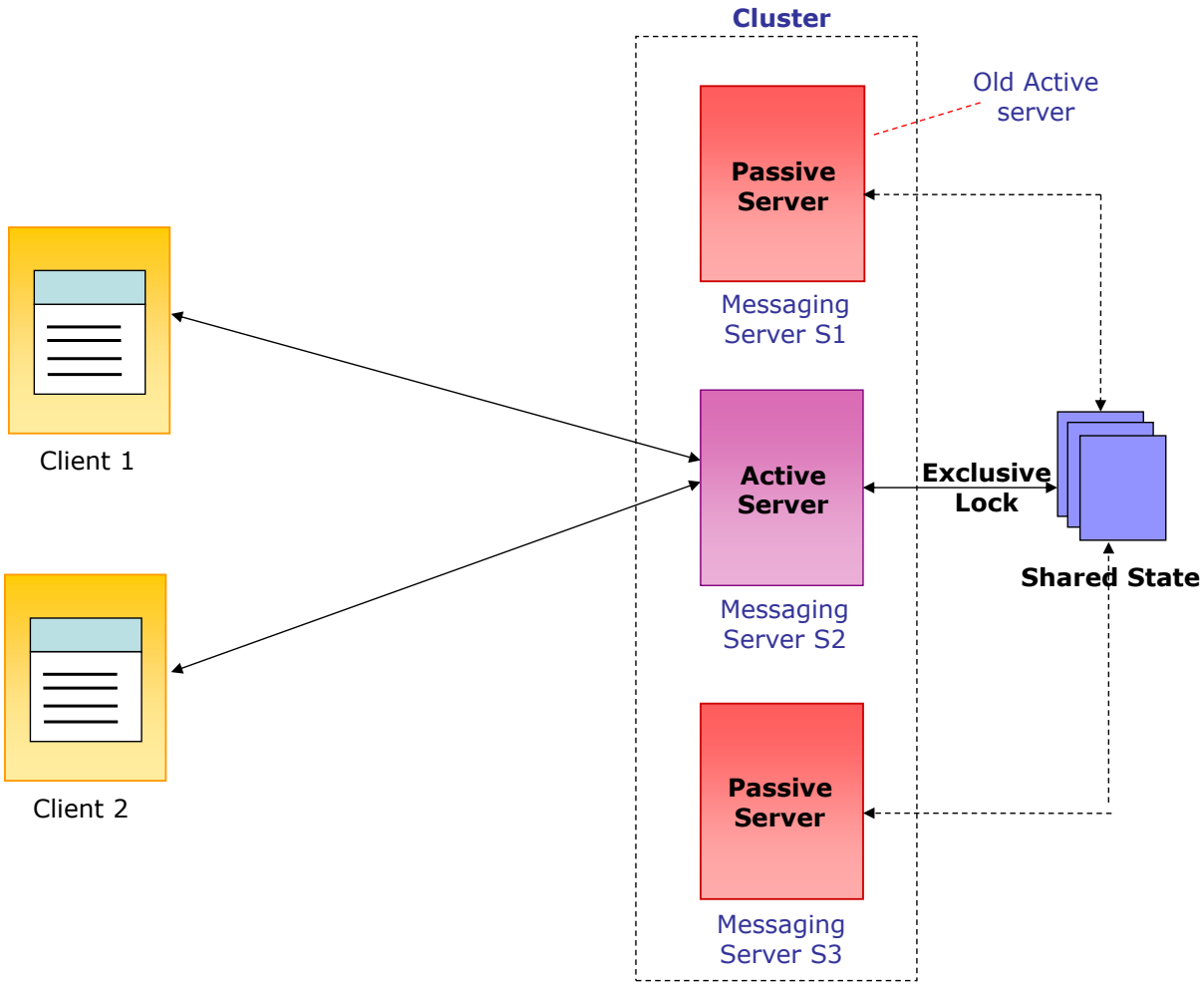

Figure: Failover Scenario and Role Swap

However, note that the NON-PERSISTENT messages are lost during failover as these messages are stored in memory of messaging server. So any critical messages should be PERSISTENT to sustain messaging server failure.

Configuration of Client for Fault Tolerance

Providing list of messaging servers URLs configured in a cluster to client applications is one of the many options available to make messaging server failure transparent to client applications. Below is just an example on how the failover configuration looks like for client applications –

failover :(tcp: //MessagingServerS1:5100, tcp://MessagingServerS2:5101, tcp://MessagingServerS3:5102)

In case connection to active server is lost, client attempts to re-connect to servers listed in failover server list one by one until it becomes successful in connecting to one of the available servers. It all depends on the Messaging vendor and its client libraries as to how multiple servers in a cluster are specified to client and how client libraries manages

reconnect mechanism with other available servers in case it detects connection failure to active server. Specifying server list and approach may differ from vendor to vendor.

Advisory Messages and Monitoring

Monitoring the overall messaging infrastructure and components is very important aspect to ensure server is performing as expected and any adverse situation is notified and corrective actions are taken on time. Advisory messages allow to monitor the events associated with messaging server, clients (consumers and producers) and destinations.

Some of the examples of these events (advisory messages) are

- Messaging server resource utilization reaching threshold (server related)
- Server to Server bridge not active (server related)
- Consumer and Producer connection start or stop (client related)
- No consumer on destination (client related)
- Destination capacity reaching threshold limit (destination related)
- Destination overflow – allowed capacity consumed (destination related)

These events get published to pre-defined advisory destinations (usually topics) which are monitored and then corrective action can be taken. Basically, there can be subscriber clients monitoring the individual advisory topic. Once subscriber gets the message it can analyze the message and take the corrective action.

Let's take an example of Server to Server Bridge monitoring. If the monitoring is enabled on bridge to monitor its active or inactive status, then if bridge becomes inactive (impacting server to server communication), then advisory event is generated and published to the pre-defined advisory topic – for example, *Advisory.serverBridge*. Subscriber client listening on this topic consumes the message and can send an email notification or SMS to administrator indicating bridge failure and suggesting action to start the Bridge. Alternatively, it can trigger the script which can automatically start/reset the bridge.

The diagram below depicts this scenario.

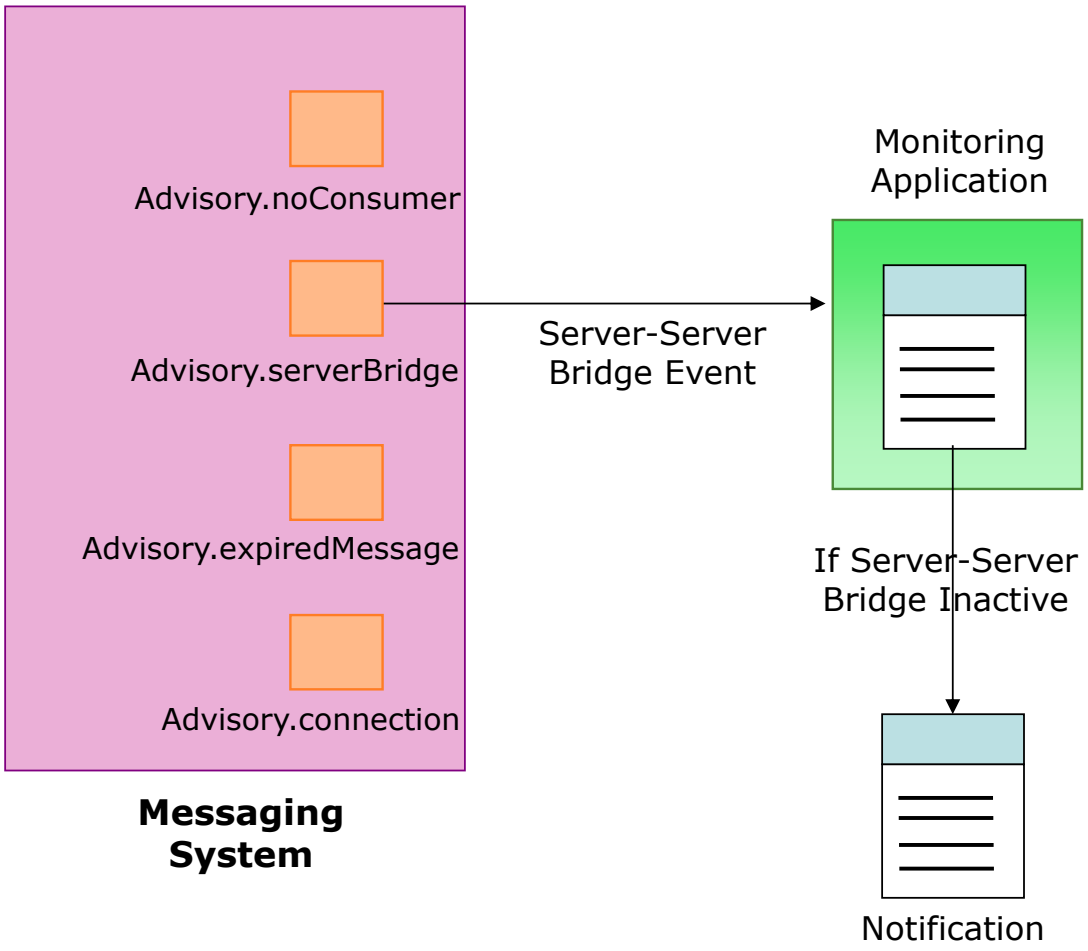

Figure: Advisory Messages and Monitoring

Similarly consumer status on destination (queue or topic) can be monitored and event can be generated if there are no consumers on the destination. Alerting this and taking corrective action of starting consumer can avoid the situation where messages are piling up in destination without processing.

Anti-Patterns and Performance Consideration

In this section, let's take a look at some of the key anti-patterns in Messaging space.

Anti-Pattern: Making most of the transactions as PERSISTENT
Impact: Performance of Messaging system and overall SLA

Not all messages/events are critical. Wherever possible, make messages NON-PERSISTENT; PERSISTENT messages impact the performance of messaging server as it involves IO operations which are expensive. The thumb rule is to mark Request-Reply messages as NON-PERSISTENT and mark Fire-and-Forget messages as PERSISTENT. Request-Reply

operations are usually data retrieval operations, even if request or reply message is lost and response is not received by requester application, it will get timeout and can re-raise the request. In Fire-and-Forget scenario, it is one way message and there is no way sender can re-send the message and hence it is important to mark these messages as PERSISTENT to ensure guaranteed delivery to receiver. Having said that, some of the one way messages can be marked as NON-PERSISTENT if they are non-critical in nature; for example, some acknowledgment/notification messages.

Anti-Pattern: Exchanging large or bulk data using Messaging system.
Impact**:** Performance of Messaging system.

Since guaranteed delivery is required while exchanging data between two applications, tendency is to look at messaging as a solution and often the size and category of data is overlooked. Further if this data needs to be exchanged or synchronized on real-time basis, messaging is considered as apt solution. Size and categorization of data are very crucial aspects. Data which is larger in size falls under *Batch data* category and doesn't come under *message* category. Though there is no hard and fast rule, messages are typically smaller in size (in Bytes and KBs) and are real-time events. Any request-reply or fire-and-forget events or publication events are always smaller in size. Using Messaging system to exchange batch or bulk data will impact memory utilization and IO operation (if PERSISTENT), thus slowing down messaging server.

To ensure guaranteed delivery and real-time synchronization of batch data or files there are alternate solutions which are more efficient in handling bulk data exchange.

Anti-Pattern: Connecting Business Critical applications to a single messaging server.
Impact**:** Impact on Business.

Most of the time, strategy for distributing business critical applications across messaging server is ignored to save the cost on messaging server and applications supporting business critical functionalities end up connecting to a single messaging server. Though messaging server setup is configured for high availability, there could be still issues which can cause messaging server downtime. For example, the issue could be related to mis-configuration which can cause messaging server to go down or mis-behavior from one of the clients in terms of connection loop or continuously sending duplicate messages that may exhaust messaging server memory and processor utilization. It is very important that business critical applications are well distributed across multiple messaging servers, factors such as volume growth, peak time load should also be taken into consideration while distributing applications across messaging servers. For example, let's take the case of Electronics product retailer whose business primarily depends on customers shopping through various channels such as online Portal, Call Center and Retail Outlets. As up-time of channels is critical to their business, it may make sense to distribute these channel applications across multiple servers so that mis-behavior of one of the servers will not cause downtime for all other channels, thus reducing risk to Business. Same is the case with other customer centric domain like Telecom, BFSI; channel applications must be distributed across multiple messaging servers to reduce business impact in case of unforeseen failure.

Anti-Pattern: Too many Server to Server Bridges for response sensitive applications.
Impact**:** Impact on response SLA and performance.

For response sensitive applications, it is very important that Service Consumer and Service Provider applications are not connected via multiple Server to Server bridges, message routing across servers and over the network can lead to significant latency. Overall response SLA must be taken into consideration while distributing provider and consumer applications across multiple messaging servers; in some cases where low latency between consumer and provider applications is the dominant non-functional requirement, it might be good idea to connect them on the same messaging server or in MESH topology. Additionally, messaging vendor may have specific messaging products/solutions to address low latency requirements, which might be worth considering.

Unit I - 'Message-Oriented Middleware'
Takeaways

Message-Oriented Middleware – Takeaways

- Messaging is the first generation application integration mechanism for integrating heterogeneous applications.
- Organizations will have use cases where applications want to exchange data in pre-agreed format and semantic, with guaranteed delivery and in reliable manner – Messaging-Oriented Middleware (MOM) is the solution for such requirements.
- Use Messaging to provide loose coupling and standards based mechanism to interfacing applications for exchanging data; this will prevent applications from impacting each-other during technology/application upgrade programs.
- Messaging system supports 2 fundamental styles of communication. Choose appropriate messaging style based on data exchange requirement
  - Point-To-Point (Queue based)
  - Publish-Subscribe (Topic based)
- Use Point-To-Point model when there is only one consumer of the data. Point-to-Point communication could be
  - Request-Reply pattern (consumer of request sending back the response)
  - Fire-and-Forget pattern (one way message)
- Use Publish-Subscribe style for broadcasting information to multiple consumers/subscribers. Publish-Subscribe communication is always one way message, from publisher to subscribers.
- Use JMS features effectively for appropriate design - CorrelationID, ReplyTo Queue, Priority, JMS Properties, Message Selector etc.
- Ensure no loss of Business critical events by using PERSISTENT Delivery Mode, Client Acknowledgement and No Expiry.
- Use JMS APIs for building client applications – sender and receiver applications. Avoid using proprietary APIs of Messaging vendor to send and receive data. Use of JMS APIs will provide abstraction from vendor specific APIs and will facilitate smooth migration to any JMS complaint messaging server with minimal impact to interfacing applications.
- Do not connect multiple critical applications to same messaging server; distribute them across multiple Messaging servers to reduce business impact. Multiple messaging servers will also be required for better handling of growing volume.
- Use Server to Server Bridge for propagating data through multiple Messaging servers.

- Focus on building federated architecture to have separation of concern, reduced business impact and scalability by design itself. Depending on the assessment, choose appropriate topology (or Mixed Topology)
  - Hub and Spoke
  - Mesh
- Ensure important non-functional aspects such as Fault Tolerance and Monitoring are taken care of.
- Avoid common messaging pitfalls such as
  - Marking most of the messages as PERSISTENT.
  - Exchanging large or bulk data using messaging.
  - Connecting multiple Business critical applications to same messaging server.
  - Too many Server to Server bridges (hops) for response sensitive applications.

About The Author

Pravin Gokhe specializes in Application Integration, SOA, BPM, Messaging Middleware and API Management space. Pravin has over 18 years of practical experience in the field of Information Technology.

Pravin has been instrumental in defining real-time integration strategies and designing large scale integration-SOA-BPM-B2Bi-API solutions for top customers from Banking, Supply Chain and Telecom domain. Pravin has worked on broad range of middleware technologies from leading vendors - TIBCO, Software AG (webMethods), Oracle (Fusion Middleware) and IBM.

Pravin has published several articles/thought papers on API, SOA, iPaaS and Integration in general. He loves to research new trends-strategies in real-time integration space including API Economy and iPaaS.